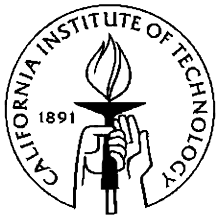
The design of high performance asynchronous circuits for the Caltech MiniMIPS processor

Thesis by

Paul Pénzes

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California, U.S.A.



1998
(Submitted May, 1998)

Contents

1 The Fetch of the Caltech MiniMIPS microprocessor

1.1 Introduction

The purpose of this report is to describe the design and implementation of an asynchronous Fetch unit used in the high-performance Caltech MiniMIPS microprocessor [6]. The Caltech MiniMIPS microprocessor was designed based on the Martin synthesis techniques [1]. The main goals of this project were to investigate new architectural issues in asynchronous processor design and to develop new techniques and tools that can meet high throughput requirements.

At the moment of writing, the chip is getting it's final touches before fabrication. Analog simulations show that in 0.6μm MOSIS SCMOS, the instruction execution frequency is expected to be approximately 280 MIPS for a power consumption of 7W at 75°C.

1.2 The FETCH and its environment

The FETCH unit is part of a distinct pipeline of the processor referred as the *fetch loop*, consisting of the FETCH (referred as PC unit in [6]), ICACHE (referred as FETCH in [6]) and DECODE. This is the loop in which the program counter is maintained. The other main pipeline is the *execution pipeline* consisting of the execution units, the register unit, and the write-back. The MIPS ISA specifies a one-instruction branch delay slot (i.e., branches do not take effect immediately, but with one instruction delay), thus the *fetch loop* operates on two tokens concurrently; effectively computing the program counter of two different instructions at a time. On the other hand, the number of tokens in the *execution pipeline* is variable.

The presence of two tokens in the *fetch loop* is relevant for understanding how the Fetch works. Consider the program sequence of three instructions `instr1; instr2; instr3`. `instr2` will be fetched and executed independently of the fact that `instr1` is a branch instruction or not. Only the fetching and execution of `instr3` is dependent on the nature of `instr1`. The program counter of the instruction following `instr2` is generated based on `instr1` and

not based on `instr2`.

The instructions executed by the MIPS R3000 could be divided from the point of view of the Fetch Unit into: regular instructions - instructions that do not explicitly affect[1] the execution flow (nops, adds, shifts), and execution flow instructions (branches) that explicitly affect execution flow. In case of a regular instruction, the program counter generated based on it is always the previous program counter incremented with four. However, the value of the program counter generated based on a branch instruction is not fully known at the time of its computation. The information is not available early enough to decide if the new program counter has to be the target of the branch (in case of a taken branch) or the previous program counter incremented (in case of a not taken branch). For a branch instruction, the register file has to send the operand(s) of the comparison (that computes the branch condition) to the FETCH. Then, the FETCH has to execute the 32-bit comparison and decide if the branch is taken or not. To compute the next program counter without a stall, this information should be available at the beginning of the FETCH cycle. However, at an early stage of the design, it turned out that it would take about one and a half cycles to compute the branch information.

1.2.1 Branch prediction

In order to avoid a stall in the processor on branches, we developed two optimizations: pre-decoding and a simple branch prediction scheme. The pre-decoding consists of overlapping the tags comparison from the extracted cache line with speculatively decoding it. [see 4 for more details]. The branch prediction scheme we used is generally know as *static branch prediction*. We assume that backward (loop) branches are *taken* and forward (if) branches are *not taken*. The intuition behind predicting backward branches as *taken* is that they are mostly loop branches. Since loops in general take several iteration, the prediction is correct for all the iterations, except the last one - when the program flow leaves the loop (statistically, the prediction is correct with about 90%). The choice for forward branches predicted as *not taken* is based on statistical measurements and the probability of correct prediction is about 58%

[1] Even a regular instruction can affect the execution flow by raising an exception

[2].

The information about the nature of the branch (forward or backward) is available early, indeed it is the 15-th bit of the instruction being decoded. Once the speculative program counter is computed, it is sent along the same path as a regular program counter. The instruction corresponding to the speculative program counter is extracted from the cache. Before it gets decoded it is matched against a filtering bit (the toss bit) computed meanwhile in the FETCH by comparing the corresponding branch operands. Note that, without prediction the new instruction would not be extracted speculatively from the cache; rather, there would be a stall till the branching information becomes available. If the prediction was correct, the newly extracted instruction goes through and it is decoded as a valid next instruction. If the prediction was incorrect, the extracted instruction gets tossed and the correct instruction is extracted from the cache on the next cycle.

To recover in case of a mispredicted program counter, the FETCH has to save the not taken program counter (while sending out the speculative program counter). In case of a mispredict, the Fetch has to resend this saved value on the next cycle. This "correction" cycle is implemented to be completely transparent from outside (except for parts of the ICACHE, i.e. the TOSS process which receives the information about the correctness of the prediction). In particular, the correctness of the exception handling mechanism is not affected. In case of a mispredicted instruction, the exception line is not sampled. It will be sampled only on the next cycle when the correct value of the program counter is dispatched.

The status of the predicted program counter is kept in the *mp* bit. Computing this bit involves a register value retrieval (an operation with high latency) and a comparison. In order to shadow the register file latency, the computation of *mp* is postponed as much as possible (within the cycle) and optimized for a low forward latency. We estimated a branch prediction rate of 65% correct. We measured the mispredict rate on dhrystone: out of 89 branches 65 were correctly predicted; higher than the average estimate. For the correctly predicted branches, the pipeline stall is expected to be about 3/8 of a cycle. It takes 8 stages from the TOSS Unit until the register file operand(s) arrive at the comparator and 2 more stages for the comparison. However, the result

of the comparison would be needed after 7 stages; thus, the difference of 3 stages. The same number of stage delays occur (as it will be explained in the slack-matching section of this chapter) on the program counter - the two delays match-up. To improve the stall on correctly predicted branches, both paths would need to be shortened. For the incorrectly predicted branches the pipeline stall is expected to be about 11/8 cycles; since it takes one more cycle to dispatch the correct program counter. Without a branch prediction, for every branch instruction we would encounter a 11/8 cycle stall. Given the estimated high hit rate and the relatively low cost of the branch prediction, it is clearly a worth while mechanism to implement.

To further exemplify how the prediction mechanism works, consider the following instruction stream:

```
instr0
instr1
if_cond_branch
instr2
instr3
instr4
instr5
```

For simplicity, assume that the prediction mechanism asserted `instr3` to be the next instruction to be fetched. We will call the toss bit the true/false value of the branch condition. Based on this bit, the speculatively fetched instruction is allowed to continue execution or is discarded. The branch prediction mechanism is shown in figure 1. The figure shows part of the decomposed Fetch Unit which will be presented in detail later; BJ, MPL and V are control processes, PCM is the process of the Fetch that computes the program counter, COMP1 and COMP2 make up the comparator for computing the jump condition and the black dots show the tokens in the system.

Initially, the program counter for `instr1` (first token) is generated with a false toss bit. At the same time `instr0` (second token) is extracted from the cache, matched against the initial toss bit (that is false) and allowed to continue execution. Next, based on `instr0` the program counter for `if_cond_branch` is computed with toss bit false, while `instr1` is extracted from the cache. Based on `instr1` the program counter for `instr2` is computed with toss

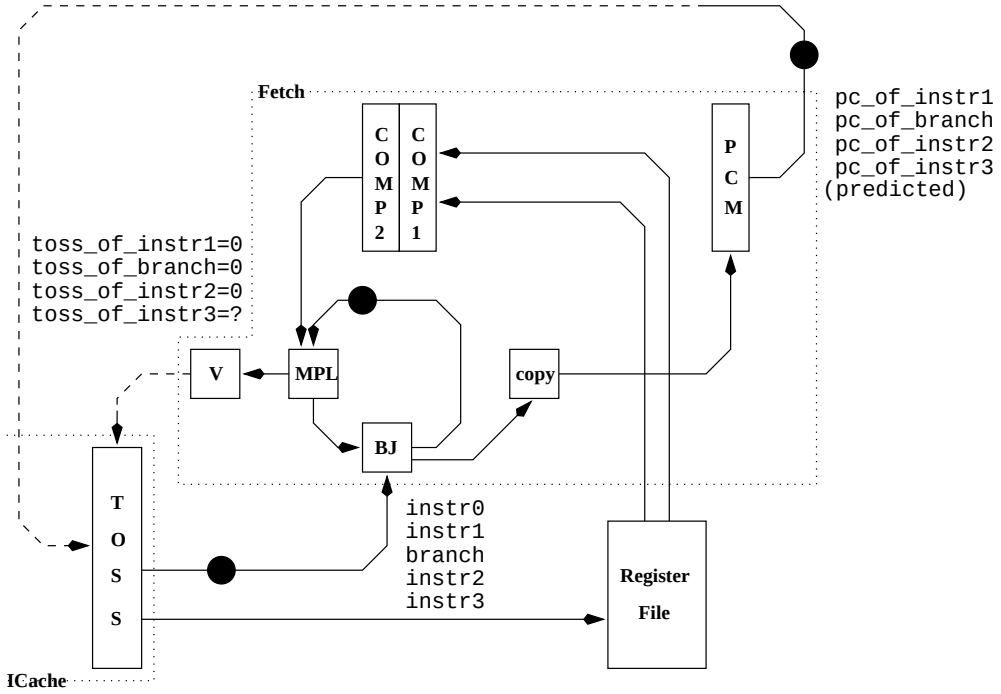


Figure 1: *Branch prediction mechanism*

bit false, while `if_cond_branch` is extracted from the cache. Now, based on `if_cond_branch` the program counter of the predicted instruction `instr3` is generated, while `instr2` is extracted from the cache. The toss bit for `instr3` is generated at the end of this cycle with an added delay of 3/8 cycles. Based on this toss bit `instr3` (just extracted from the cache core) passes through or gets discarded by the TOSS Unit. In case that `instr3` is tossed, the FETCH sends a new, (now correct) program counter in the next cycle. This extra send causes an additional one cycle stall in the *fetch loop*.

1.3 Introducing the Fetch

Consider the following CHP program for a Fetch:

$$
\begin{aligned}
FETCH\_a \equiv\ & pc := init\_pc; \\
& *[F?ins;\ \ pc2 := pc + 4; \\
& \quad [ins.b \longrightarrow BRIMM?brimm,\ \ A?a,\ \ B?b; \\
& \qquad\qquad target := pc + 4{*}brimm,\ \ realj := \underline{ccode}(a, b) \\
& \quad [] \neg ins.b \longrightarrow skip \\
& \quad]; \\
& \quad [ins.b \ \wedge\ realj \ \longrightarrow\ pc := target \\
& \quad [] \neg ins.b \ \vee\ \neg realj \ \longrightarrow\ pc := pc2 \\
& \quad]; \\
& \quad PC!pc; \\
&]
\end{aligned}
$$

This process reads in a control token from channel F and produces a program counter (*pc*) on channel PC. The control token is read into the variable *ins*. This variable has its field *.b* set to true in case of a branch and to false otherwise. Later on, variable *ins* will be extended with more fields. After reading in the control token, the previous program counter gets incremented and assigned to variable *pc2*. In case of a branch instruction, the branch immediate value is read in from channel BRIMM, while the operands of the branch comparison are read in from the register file on channels A and B. If the input token is not a branch instruction, the next program counter sent on PC will be the previous program counter incremented by four (*pc2*). If the input token is a branch instruction, the target of the branch (*target*) and the jump condition (*realj*) are computed. The jump condition is computed by the function *ccode* by comparing variables *a* and *b*. If the jump condition is true, the branch will be taken and the newly computed *target* will be sent out as the new program counter. If the jump is not taken, the previous program counter incremented by four (*pc2*) is sent out on channel PC. Note that since there are two tokens in the *fetch loop*, the input token refers to the program counter fetched two cycles before (and not to the one fetched in the previous cycle).

The main shortcoming of this CHP program is that *realj* cannot be computed early enough to avoid a stall in the pipeline. Computing *realj* involves reading in the comparison operands from the register file (which has a high latency) and comparing them against each other. We estimated that computing *realj* would take about one and a half cycles. Instead of stalling the pipeline

for one and a half cycles, in the next CHP the program counter is generated speculatively without waiting for *realj*. By the time the new (speculative) instruction is brought up from the cache, the value of *realj* is known. If the speculation was correct nothing special happens and the speculative instruction is allowed execution. However, if the speculation was incorrect, ICACHE has to cancel the fetched instruction. The FETCH communicates on channel TOSS to the ICACHE the status of the speculative instruction. On the other hand, in case of an incorrect speculation the FETCH should be able to recover and generate the correct *pc*. Thus, it has to save the not taken alternative of the program counter (*nottaken*). Furthermore, we will predict the backward branches as taken and the forward branches as not taken. There is one more bit of information needed to tell the nature of the branch (input channel FB). Incrementing the previous *pc* by four could be done in parallel with reading in the control token. The new CHP that incorporates the branch prediction mechanism and the parallel *pc* incrementation is:

$$
\begin{array}{l}
FETCH\_b \equiv \;\; pc := init\_pc; \\
\quad *[F?ins, \;\; FB?sign, \;\; pc2 := pc + 4; \\
\quad\quad [ins.b \longrightarrow BRIMM?brimm; target := pc + 4{*}brimm, j := sign \\
\quad\quad [\!][\neg ins.b \;\; \longrightarrow \;\; j\downarrow]; \\
\quad\quad [j \;\; \longrightarrow \;\; pc := target, nottaken := pc2 \\
\quad\quad [\!][\neg j \;\; \longrightarrow \;\; pc := pc2, nottaken := target \\
\quad\quad]; \\
\quad\quad PC!pc, \\
\quad\quad [ins.b \;\; \longrightarrow \;\; A?a, \;\; B?b; \;\; realj := \underline{ccode}(a, b) \\
\quad\quad [\!][\neg ins.b \;\; \longrightarrow \;\; skip]; \\
\quad\quad TOSS!(j \neq \;\; realj); \\
\quad\quad [j \neq \;\; realj \;\; \longrightarrow \;\; PC!nottaken, \;\; TOSS!false \\
\quad\quad [\!][j = realj \;\; \longrightarrow \;\; skip \\
\quad\quad] \\
\quad]
\end{array}
$$

We notice that in case of a mispredicted program counter, there will be one more communication on channel PC and TOSS in that cycle. To conditionally execute more than one send on the same channel during one cycle could be difficult to implement. Instead, we introduce a mispredict variable *mp* to

distinguish a normal cycle from a mispredict cycle. We transform our CHP into:

$$
\begin{array}{l}
FETCH\_c \equiv \;\; pc := init\_pc, \;\; mp\downarrow; \\
\quad *[[\neg mp \longrightarrow F?ins, \;\; FB?sign, \;\; pc2 := pc + 4 [] mp \longrightarrow skip]; \\
\quad\quad [\neg mp \wedge ins.b \longrightarrow BRIMM?brimm; target := pc + 4*brimm, j := sign \\
\quad\quad [] \neg ins.b \;\vee\; mp \;\longrightarrow\; j\downarrow]; \\
\quad\quad [\neg mp \;\wedge\; j \;\longrightarrow\; pc := target, nottaken := pc2 \\
\quad\quad [] \neg mp \;\wedge\; \neg j \;\longrightarrow\; pc := pc2, nottaken := target \\
\quad\quad [] mp \;\longrightarrow\; pc := nottaken \;]; \\
\quad\quad PC!pc, \\
\quad\quad [\neg mp \wedge ins.b \longrightarrow realj := \underline{ccode}(A?, B?); mp := realj \neq \; j \\
\quad\quad [] \neg ins.b \;\vee\; mp \;\longrightarrow\; mp\downarrow]; \\
\quad\quad TOSS!mp \\
\quad]
\end{array}
$$

To this previous CHP we add the exception handling mechanism. In case of an exception, the new program counter is set to a constant value ($ex\_pc - 4$). The event that the exception was observed by the Fetch and that the WriteBack can start executing instructions again is signaled via channel VA. On a mispredict cycle, the exception channel is not probed, since on a true mp only the sends on channels PC and $TOSS$ are suppose to happen.

$$
\begin{array}{l}
FETCH\_d \equiv \;\; pc := init\_pc, \;\; mp\downarrow, \;\; va\downarrow; \\
\quad *[[\neg mp \longrightarrow F?ins, \;\; FB?sign, \;\; pc2 := pc + 4 [] mp \longrightarrow skip]; \\
\quad\quad [\neg mp \wedge ins.b \longrightarrow BRIMM?brimm; target := pc + 4*brimm, j := sign \\
\quad\quad [] \neg ins.b \vee mp \;\longrightarrow\; j\downarrow \;]; \\
\quad\quad [\neg mp \longrightarrow [\overline{EX} \longrightarrow EX, va\uparrow, pc := ex\_pc - 4 \\
\quad\quad\quad\quad\quad\quad | \neg \overline{EX} \;\longrightarrow\; [j \wedge \neg va \longrightarrow pc := target, nottaken := pc2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [] \neg j \vee va \longrightarrow pc := pc2, nottaken := target \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad]; \;\; va\downarrow] \\
\quad\quad [] mp \;\longrightarrow\; pc := nottaken \;]; \\
\quad\quad PC!pc, \;\; VA!va, \\
\quad\quad [\neg mp \wedge ins.b \longrightarrow realj := \underline{ccode}(A?, B?); mp := (realj \neq \; j) \wedge \neg va \\
\quad\quad [] \neg ins.b \;\vee\; mp \;\longrightarrow\; mp\downarrow \\
\quad\quad]; \;\; TOSS!mp \\
\quad]
\end{array}
$$

There are a few more details that need to be added to FETCH_d to get the final high-level CHP of the Fetch that we implemented. First, F is a one-of-four channel; thus there are 4 types of branch instructions: no-branch ($ins.n$), jump immediate ($ins.i$), jump register ($ins.r$) and regular branch ($ins.b$). Furthermore, in case of a regular branch, there are 6 types of branches ($\geq$, $\leq$, etc). This information is sent to the Fetch on channel F2. Since the Fetch should be able to execute branch-and-link instructions, too; it has to be able to send the current pc to the register file. This is done on channel C, while the link information is sent to the Fetch on channel G (assigned to variable bl with fields $bl.0$, $bl.1$ and $bl.2$). In case of an exception, the exception handler has to know if the instruction raising an exception was in a branch delay slot. To communicate this to the CP0 unit we add an output channel to the Fetch called BD. Now, we are ready to present the top level CHP of our Fetch.

$$
\begin{array}{l}
FETCH \equiv \; PC!init\_pc,\;\; VA!false,\;\; TOSS!false,\;\; pc := init\_pc,\;\; va{\downarrow},\;\; va2{\downarrow},\;\; mp{\downarrow}; \\
\quad *[[\neg mp \longrightarrow F?ins, G?bl, FB?sign, F2?bc, pc2 := pc + 4 [] mp \longrightarrow skip]; \\
\quad\quad [\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4*(IMHI?, JIMM?)), j{\uparrow}, bd{\uparrow} \\
\quad\quad [] \neg mp \wedge ins.r \longrightarrow A?target, j{\uparrow}, bd{\uparrow} \\
\quad\quad [] \neg mp \wedge ins.b \longrightarrow BRIMM?brimm; target := pc + 4*brimm, j := sign, bd{\uparrow} \\
\quad\quad [] \underline{else} \longrightarrow j{\downarrow}, bd{\downarrow} \\
\quad\quad], \\
\quad\quad [\neg mp \;\; \wedge \;\; bl.2 \longrightarrow \;\; C!pc2 \\
\quad\quad [] \underline{else} \;\; \longrightarrow \;\; skip]; \\
\quad\quad [\neg mp \longrightarrow [\;\; \overline{EX} \longrightarrow EX, va{\uparrow}, va2{\uparrow}, pc := ex\_pc - 4 \\
\quad\quad\quad\quad\quad\quad | \neg \overline{EX} \longrightarrow [j \wedge \neg va \longrightarrow pc := target, nottaken := pc2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [] \neg j \vee va \longrightarrow pc := pc2, nottaken := target \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad]; \;\; va2 := va; va{\downarrow} \\
\quad\quad\quad\quad\quad\quad] \\
\quad\quad [] mp \longrightarrow pc := nottaken \\
\quad\quad]; \\
\quad\quad PC!pc, \;\; VA!va, \;\; BD!bd;
\end{array}
$$

$$
\begin{array}{l}
[\neg mp \wedge ins.b \longrightarrow [bl.0 \vee bl.2 \longrightarrow realj := \underline{ccode}(bc, (A?), 0) \\
\qquad\qquad\qquad\qquad [] bl.1 \longrightarrow realj := \underline{ccode}(bc, (A?), (B?)) \\
\qquad\qquad\qquad\qquad]; \;\; mp \;\; := \;\; (realj \neq \;\; j \wedge \neg va2) \\
\quad [] \underline{else} \longrightarrow mp\downarrow \\
\quad]; \\
\quad TOSS!mp \\
]
\end{array}
$$

1.4 Specification

In the previous section we started with a simple Fetch unit and added the desired properties step-by-step. The reason for doing so was to make it easier for the reader to parse the final CHP of the Fetch unit. However, the desired properties were introduced in a different way. When we designed the Caltech MiniMIPS we specified a high-level sequential CHP program, one that met, with a few exceptions (no TLB, no unalign word accesses) the spec of the MIPS R3000 as described in [5]. After several steps of decomposition and refinement, through which pipelining and concurrency was introduced, we factored out the high-level sequential CHP for the FETCH unit.

The FETCH unit computes the next program-counter in case of a nop ($ins.n$), branch ($ins.b$), jump immediate ($ins.i$), jump register ($ins.r$) or exception. It receives most of its inputs from the TOSS unit on channels F (2 bit quad-rail, passive), G (one-of-three, passive), FB (dual-rail, passive, copy of bit 15 of the instruction) and F2 (6 bit quad-rail, passive, bits 29 to 26, 17 and 16 of the instruction). In case of an exception the WRITEBACK sends and exception signal on channel EX (dataless, passive). In case of a branch, the register file sends the operand(s) of the program counter comparison on channel(s) A (32 bit quad-rail, passive) (and B (32 bit quad-rail, passive)) and the immediate bus sends the operand on channel BRIMM (16 bit quad-rail, passive). In case of a jump immediate the immediate bus sends the operands on channels JIMM (16 bit quad-rail, passive) and IMHI (10 bit quad-rail, passive). Most of the outputs are sent to ICACHE on channels PC (32 bit quad-rail, active), VA (dual-rail, active) and TOSS (dual-rail, active). The CP0 unit receives the branch-delay bit on the BD channel (dual-rail, active). In case of a link register, the register file receives the value of the program-counter on

channel C (32 bit quad-rail, active).

1.5 Decomposition

1.5.1 Introduction

The goal of this section is to show how a complex CHP as that of the FETCH could be efficiently implemented. We will present both the available decomposition rules and the particular way in which we apply them to a given problem. As presented before, the FETCH is part of the critical *fetch loop*. Because this loop was predicted to have high forward latency we could afford only a few pipeline stages on the forward path of this unit. This put another constraint on our choices to decompose the high level CHP.

There are at least two reasons why we do process decomposition. First, as written there is no direct way available to implement the initial CHP. Secondly, along the process decomposition we can introduce both pipelining and concurrency to the initial sequential process. This can result in a more efficient implementation. Some of the necessary and sufficient conditions for the correctness of these transformations could be found in [3].

The main (simplified) template for an implementable process is

$$
\begin{array}{l}
*[C?c; \\
\quad [c=0 \longrightarrow A?a; B!\underline{f}(a) \\
\quad [\!] c=1 \longrightarrow A?a; B!\underline{g}(a) \\
\quad] \\
\;]
\end{array}
$$

where each channel communication could be a set of conditional channel communications. We try to decompose and write our processes in such a way that they have the structure of this template. We implement this template in general, with precharge half-buffering, hybrid-buffering or precharge full-buffering [7].

The main strategy in our decomposition is to factor out, step-by-step, independent modular processes from the initial CHP. In this way we introduce pipelining and in some cases concurrency.

Consider the following CHP fragment *...; m:= f(n); ....* If we decide to factor out the computation of m in a different process, we substitute the use

of m in the initial process with a receive action $A?m$ and the computation $f(n)$ with a send action $B!n$. The factored out process will be $*[B?n;A!f(n)]$ while the initial CHP transforms to $...; B!n,A?m; ....$ Action $B!$ and $A?$ are ordered by the factored out process; so, they do not need extra synchronization in the initial process. Furthermore, $B!$ could be moved back in the CHP body until the last statement that sets n; while $A?m$ could be moved forward in the CHP body until the first statement that uses m. If buffering is added on channels A and B, the factored out process and the initial one could execute concurrently.

1.5.2 FETCH decomposition

In the first step of process decomposition, we rearrange the *selection* statements of the initial CHP, such that they are not nested. We **introduce variable** e as the result of the exception line sampling and variable $b := \neg mp \& ins.b$ to decide if we recompute or not *realj*. We move the exception line sampling at the beginning of the loop. The correctness of this transformation follows from the slack elastic property of the exception handling mechanism [3].

$$
\begin{aligned}
FETCH \equiv\ & PC!\underline{init\_pc},\ VA!false,\ TOSS!false,\ va{\downarrow},\ va2{\downarrow},\ mp{\downarrow};\\
& *[[\neg mp \longrightarrow F?ins, G?bl, FB?sign, F2?bc, pc2 := pc+4,\\
& \qquad [\overline{EX} \longrightarrow EX,\ e{\uparrow}\ |\neg\overline{EX} \longrightarrow\ e{\downarrow}]\\
& \quad [] mp \longrightarrow skip];\\
& \quad [\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4*(IMHI?, JIMM?)), j{\uparrow}, bd{\uparrow}, b{\downarrow}\\
& \quad [] \neg mp \wedge ins.r \longrightarrow A?target, j{\uparrow}, bd{\uparrow}, b{\downarrow}\\
& \quad [] \neg mp \wedge ins.b \longrightarrow BRIMM?brimm;\\
& \qquad\qquad target := pc + 4*brimm, j := sign, bd{\uparrow}, b{\uparrow}\\
& \quad [] \underline{else} \longrightarrow j{\downarrow}, bd{\downarrow}, b{\downarrow}\\
& \quad],\\
& \quad [\neg mp \ \wedge\ bl.2 \longrightarrow\ C!pc2\ []\ \underline{else}\ \longrightarrow\ skip];\\
& \quad [\neg mp \wedge e \longrightarrow va{\uparrow}, va2{\uparrow}, pc := ex\_pc - 4\\
& \quad [] \neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2, va2 := va; va{\downarrow}\\
& \quad [] \neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target, va2 := va; va{\downarrow}\\
& \quad [] mp \longrightarrow pc := nottaken\\
& \quad];
\end{aligned}
$$

$PC!pc,\ \ VA!va,\ \ BD!bd;$

$[b \wedge (bl.0 \vee bl.2) \longrightarrow realj := \underline{ccode}(bc, (A?), 0); mp := (realj \neq\ \ j) \wedge \neg va2$

$[]b \wedge bl.1 \longrightarrow realj := \underline{ccode}(bc, (A?), (B?)); mp := (realj \neq\ \ j) \wedge \neg va2$

$[]\neg b \longrightarrow mp\downarrow$

$];$

$TOSS!mp$

$]$

We **factor out** the *realj* computation, together with all accesses on channel A and B into process CCODE. This new process needs parts of *ins* and *bl* from the original process. We encode this information into variable *c3* and send it to process CCODE on channel CCODE1. The result of the comparison (*realj*) is sent back from process CCODE on channel CCODE2. Channel A was used to receive the branch operand in case of a branch and the jump address in case of a jump register. Once we factored out channel A; in case of a branch, the branch operand is used in the CCODE unit to compute *realj*; however, the jump address received on channel A is still used in the original process. For this reason, we will need to add an extra "bypass" channel (CCODE3) from process CCODE to the original process on which the jump address is sent on.

We explained earlier why, in case of a branch, computing the *realj* with a low forward latency is very important. As written, computing the *realj* would take at least three stages once the operand(s) from the register file are present (two stages for the comparator and one stage to compute *realj* given the result of the comparison). Then, it would take one more stage to combine the *realj* into the control of the *pc* computation. We decided to combine the last two of these stages into one (thus eliminating one stage) and leave it in the original program; while isolating a 32 bit comparator with an auxiliary bypass channel into process CCODE. Now; in case of a branch, process CCODE does not compute *realj*, it computes if the operand from channel A is equal with the operand from channel B (or 0; depending on the type of branch) and the sign of the operand from channel A. Based on this information and the branch type, the original process computes (using function $\underline{f}$) if the branch was correctly predicted or it was mispredicted.

$FETCH \equiv CCODE\ \ ||\ \ FETCH1$

$$
\begin{array}{l}
CCODE \equiv *[\,[CCODE1?c3;\\
\qquad [c3.jr \;\longrightarrow\; CCODE3!(A?)\\
\qquad [\!]\, c3.branch1 \;\longrightarrow\; A?a,\;\; CCODE2!(a=0, a<0)\\
\qquad [\!]\, c3.branch2 \;\longrightarrow\; A?a,\;\; B?b;\;\; CCODE2!(a=b, a<0)\\
\qquad]\\
\quad]
\end{array}
$$

$$
\begin{array}{l}
FETCH1 \equiv \;\; PC!\underline{init\_pc},\;\; VA!false,\;\; TOSS!false,\;\; va{\downarrow},\;\; va2{\downarrow},\;\; mp{\downarrow};\\
\qquad *[[\neg mp \longrightarrow F?ins, G?bl, FB?sign, F2?bc, pc2 := pc+4,\\
\qquad\qquad [\overline{EX} \longrightarrow EX,\;\; e{\uparrow}\;\; [\!]\neg\overline{EX} \longrightarrow\;\; e{\downarrow}]\\
\qquad\quad [\!]\, mp \longrightarrow skip\\
\qquad\quad];\\
\\
\qquad [\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4*(IMHI?, JIMM?)), j{\uparrow}, bd{\uparrow}, b{\downarrow}\\
\qquad [\!]\, \neg mp \wedge ins.r \longrightarrow CCODE1!c3.jr, CCODE3?target, j{\uparrow}, bd{\uparrow}, b{\downarrow}\\
\qquad [\!]\, \neg mp \wedge ins.b \longrightarrow BRIMM?brimm, j := sign, bd{\uparrow}, b{\uparrow},\\
\qquad\quad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1 [\!]\, bl.1 \longrightarrow CCODE1!c3.branch2];\\
\qquad\quad target := pc + 4*brimm\\
\qquad [\!]\, \underline{else} \longrightarrow j{\downarrow}, bd{\downarrow}, b{\downarrow}],\\
\qquad [\neg mp \;\wedge\; bl.2 \longrightarrow\;\; C!pc2 \;[\!]\; \underline{else} \;\longrightarrow\; skip];\\
\qquad [\neg mp \wedge e \longrightarrow va{\uparrow}, va2{\uparrow}, pc := ex\_pc - 4\\
\qquad [\!]\, \neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2, va2 := va; va{\downarrow}\\
\qquad [\!]\, \neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target, va2 := va; va{\downarrow}\\
\qquad [\!]\, mp \longrightarrow pc := nottaken];\\
\qquad PC!pc,\;\; VA!va,\;\; BD!bd;\\
\qquad [b \longrightarrow CCODE2?(eq, sg);\;\; mp := \neg va2 \;\wedge\; \underline{f}(bc, j, eq, sg)\\
\qquad [\!]\, \neg b \longrightarrow mp{\downarrow}];\\
\qquad TOSS!mp;\\
\quad]
\end{array}
$$

As a next decomposition step we **unroll the loop** for part of the first cycle to incorporate the initial send on channel TOSS into the loop. We want to transform $P_B; *[P_A; P_B]$ into $*[P_B; P_A]$. Consider

$$
\begin{array}{l}
P_A \equiv [\neg mp \longrightarrow \;\; ... \;\; [\!]\, mp \longrightarrow \;\; ...];\\
\qquad ...\\
\qquad PC!pc,\;\; VA!va,\;\; BD!bd;
\end{array}
$$

$$P_B \equiv [b \longrightarrow \ ...;\ \ mp := ...[]\neg b \longrightarrow mp{\downarrow}];$$
$$\quad TOSS!mp;$$

With this definitions we have:

$$FETCH1 \equiv PC!\underline{init\_pc},\ \ VA!false,\ \ TOSS!false,\ \ va{\downarrow},\ \ va2{\downarrow},\ \ mp{\downarrow};$$
$$*[\ P_A;\ \ P_B]$$

If we initialize $b\downarrow$ we could write:

$$FETCH1 \equiv PC!\underline{init\_pc},\ \ VA!false,\ \ va{\downarrow},\ \ va2{\downarrow},\ \ b{\downarrow};$$
$$[false \longrightarrow \ ...;\ \ mp := ...[]true \longrightarrow mp{\downarrow}];$$
$$TOSS!false;$$
$$*[\ P_A;\ \ P_B]$$

that is:

$$FETCH1 \equiv PC!\underline{init\_pc},\ \ VA!false,\ \ va{\downarrow},\ \ va2{\downarrow},\ \ b{\downarrow};$$
$$P_B;$$
$$*[\ P_A;\ \ P_B]$$

or

$$FETCH1 \equiv PC!\underline{init\_pc},\ \ VA!false,\ \ va{\downarrow},\ \ va2{\downarrow},\ \ b{\downarrow};$$
$$*[\ P_B;\ \ P_A]$$

With this transformation, the equivalent FETCH1 CHP is:

$$FETCH1 \equiv PC!\underline{init\_pc},\ \ VA!false,\ \ va{\downarrow},\ \ va2{\downarrow},\ \ b{\downarrow};$$
$$*[[b \longrightarrow CCODE2?(eq, sg);\ \ mp\ :=\ \neg va2\ \wedge\ \underline{f}(bc, j, eq, sg)$$
$$\quad []\neg b \longrightarrow mp{\downarrow}];$$
$$\quad TOSS!mp;$$
$$\quad [\neg mp \longrightarrow F?ins, G?bl, FB?sign, F2?bc, pc2 := pc + 4,$$
$$\qquad\qquad [\overline{EX} \longrightarrow EX,\ \ e{\uparrow}\ |\neg\overline{EX} \longrightarrow\ \ e{\downarrow}]$$
$$\quad []mp \longrightarrow skip];$$

$[\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4{*}(IMHI?, JIMM?)), j\uparrow, bd\uparrow, b\downarrow$

$[\!]\neg mp \wedge ins.r \longrightarrow CCODE1!c3.jr, CCODE3?target, j\uparrow, bd\uparrow, b\downarrow$

$[\!]\neg mp \wedge ins.b \longrightarrow BRIMM?brimm, j := sign, bd\uparrow, b\uparrow,$

$\quad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1 [\!] bl.1 \longrightarrow CCODE1!c3.branch2];$

$\quad target := pc + 4{*}brimm$

$[\!]\underline{else} \longrightarrow j\downarrow, bd\downarrow, b\downarrow],$

$[\neg mp \wedge bl.2 \longrightarrow C!pc2 [\!] \underline{else} \longrightarrow skip];$

$[\neg mp \wedge e \longrightarrow va\uparrow, va2\uparrow, pc := ex\_pc - 4$

$[\!]\neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2, va2 := va; va\downarrow$

$[\!]\neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target, va2 := va; va\downarrow$

$[\!] mp \longrightarrow pc := nottaken];$

$PC!pc, \quad VA!va, \quad BD!bd;$

$]$

We move the exception sampling again, even earlier in the cycle (the correctness of this step was argued before) and we **factor out** the *mp* computation together with the exception sampling from FETCH1 into process MPE. The factored out process will need the values of *b*, *bc*, *va2* and *j* from the original process; these values will be sent to MPE on channels B, BC, VA2old and J. MPE will send back the values of mispredict (*mp*) and exception (*e*) on channels MP respectively E.

$FETCH1 \equiv FETCH2 \;||\; MPE$

$MPE \equiv *[B?b, BC?bc, VA2old?va2, J?j;$

$\quad [b \longrightarrow CCODE2?(eq, sg); \quad mp := \neg va2 \wedge \underline{f}(bc, j, eq, sg)$

$\quad [\!]\neg b \longrightarrow mp\downarrow$

$\quad]; \quad MP!mp,$

$\quad [\neg mp \longrightarrow [\overline{EX} \longrightarrow EX, \quad e\uparrow \;|\neg \overline{EX} \longrightarrow e\downarrow] [\!] mp \longrightarrow skip];$

$\quad E!e$

$]$

$$
\begin{aligned}
FETCH2 \equiv\ & PC!\underline{init\_pc},\ \ VA!false,\ \ va\downarrow,\ \ va2\downarrow, \\
& B!false,\ \ BC!\underline{anything},\ \ VA2old!\underline{anything},\ \ J!\underline{anything} \\
& *[MP?mp,\ \ E?e; \\
& \quad TOSS!mp; \\
& \quad [\neg mp \longrightarrow F?ins, G?bl, FB?sign, F2?bc, pc2 := pc + 4 \\
& \quad [] mp \longrightarrow skip \\
& \quad]; \\
& \quad [\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4*(IMHI?, JIMM?)), j\uparrow, bd\uparrow, b\downarrow \\
& \quad [] \neg mp \wedge ins.r \longrightarrow CCODE1!c3.jr, CCODE3?target, j\uparrow, bd\uparrow, b\downarrow \\
& \quad [] \neg mp \wedge ins.b \longrightarrow BRIMM?brimm, j := sign, bd\uparrow, b\uparrow, \\
& \qquad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1 [] bl.1 \longrightarrow CCODE1!c3.branch2]; \\
& \qquad target := pc + 4*brimm \\
& \quad [] \underline{else} \longrightarrow j\downarrow, bd\downarrow, b\downarrow \\
& \quad], \\
& \quad [\neg mp \ \wedge \ bl.2 \longrightarrow \ C!pc2 \ [] \ \underline{else} \ \longrightarrow \ skip]; \\
& \quad [\neg mp \wedge e \longrightarrow va\uparrow, va2\uparrow, pc := ex\_pc - 4 \\
& \quad [] \neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2, va2 := va; va\downarrow \\
& \quad [] \neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target, va2 := va; va\downarrow \\
& \quad [] mp \longrightarrow pc := nottaken \\
& \quad]; \\
& \quad PC!pc,\ \ VA!va,\ \ BD!bd,\ \ B!b,\ \ BC!bc,\ \ VA2old!va2,\ \ J!j \\
&]
\end{aligned}
$$

The value of *bc* does not get used in process FETCH2; rather, it is passed on to MPE. For this reason, we would like to **factor out** the communication on channels F2 and BC from FETCH2 into a separate process IJ. Furthermore, in the same process we would like to do some precomputation on channels B, BC and J to simplify the logic in process MPE. The value of j and b could be combined in process FETCH2. We will merge channels J and B into channel BB. The communication on F2 happens if $\neg mp$, while the communication on BC happens unconditionally. Process IJ should know the value of *mp*. We notice that the combination *b=true* and *j=false* is not valid (since if there was a branch, the jump bit gets set, too); thus, we can use this combination to encode *mp=true*. The value read in on channel F2 is bits 16 and 26 through 28 of the instruction being executed. We will decode this token into the different

types of branch instructions **factor**ing **out** process T. The output of process T is sent to process IJ on channel NBC. The result of this decoding will be combined with the branch prediction in process IJ using function $\underline{comb}(bc) = \{(bc.0, jb.1); (bc.1, jb.0); (bc.2, jb.5); (bc.5, jb.2); (bc.3, jb.4); (bc.4, jb.3)\}$. The output of this process will be the 1-of-7 channel JB. This channel tells process MPE if there is a branch; and if so, what kind of branch it is. Thus, the input channels B, BC and J of process MPL will be combined into channel JB. Instead of initializing the inputs of process IJ, we push-through the initial token, and initialize the outputs of process IJ; thus replacing $BB!\underline{no\_branch}, BC!\underline{anything}$ in process FETCH3 with $JB!0$ in process IJ.

$$FETCH2 \equiv FETCH3 \;||\; IJ \;||\; T$$

$$\begin{aligned}
FETCH3 \equiv\; & PC!\underline{init_pc},\;\; VA!false,\;\; va{\downarrow},\;\; va2{\downarrow},\;\; VA2old!\underline{anything} \\
& *[MP?mp,\;\; E?e; \\
& \quad TOSS!mp; \\
& \quad [\neg mp \longrightarrow F?ins, G?bl, FB?sign, pc2 := pc + 4 \\
& \quad [] mp \longrightarrow skip]; \\
& \quad [\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4{*}(IMHI?, JIMM?)), bd{\uparrow}, bj := \underline{no_branch} \\
& \quad [] \neg mp \wedge ins.r \longrightarrow CCODE1!c3.jr, CCODE3?target, bd{\uparrow}, bj := \underline{no_branch} \\
& \quad [] \neg mp \wedge ins.b \longrightarrow BRIMM?brimm, bd{\uparrow}, \\
& \qquad [sign \longrightarrow bj := \underline{taken_branch} [] \neg sign \longrightarrow bj := \underline{not_taken_branch}] \\
& \qquad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1 [] bl.1 \longrightarrow CCODE1!c3.branch2]; \\
& \qquad target := pc + 4{*}brimm \\
& \quad [] \neg mp \wedge ins.e \longrightarrow bd{\downarrow}, bj := \underline{no_branch} \\
& \quad [] mp \longrightarrow bd{\downarrow}, bj := \underline{mispredict}], \\
& \quad [\neg mp \;\wedge\; bl.2 \longrightarrow\; C!pc2 \;[]\; \underline{else} \;\longrightarrow\; skip]; \\
& \quad [\neg mp \wedge e \longrightarrow va{\uparrow}, va2{\uparrow}, pc := ex_pc - 4 \\
& \quad [] \neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2, va2 := va; va{\downarrow} \\
& \quad [] \neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target, va2 := va; va{\downarrow} \\
& \quad [] mp \longrightarrow pc := nottaken]; \\
& \quad PC!pc,\;\; VA!va,\;\; BD!bd,\;\; BB!bj,\;\; VA2old!va2 \\
&]
\end{aligned}$$

$$
\begin{array}{l}
IJ \equiv JB!0; \\
\quad *[BB?b; \\
\quad\quad [b = \underline{not\_taken\_branch} \longrightarrow NBC?bc;\ JB!bc \\
\quad\quad [] b = \underline{taken\_branch} \longrightarrow NBC?bc;\ JB!\underline{comb}(bc) \\
\quad\quad [] b = \underline{no\_branch} \longrightarrow NBC?bc;\ JB!0 \\
\quad\quad [] b = \underline{mispredict} \longrightarrow JB!0 \\
\quad\quad] \\
\quad]
\end{array}
$$

$$
\begin{array}{l}
T \equiv *[F2?(k := i_{28..26}, i_{16}); \\
\quad\quad [k = 0 \ \vee \ k = 2 \ \vee \ k = 3 \longrightarrow NBC!5 \\
\quad\quad [] k = 1 \longrightarrow [\neg i_{16} \longrightarrow NBC!5 \ [] \ i_{16} \longrightarrow NBC!2] \\
\quad\quad [] k = 4 \longrightarrow NBC!0 \\
\quad\quad [] k = 5 \longrightarrow NBC!1 \\
\quad\quad [] k = 6 \longrightarrow NBC!4 \\
\quad\quad [] k = 7 \longrightarrow NBC!3 \\
\quad\quad] \\
\quad]
\end{array}
$$

Correspondingly, the MPE process changes to

$$
\begin{array}{l}
MPE \equiv *[JB?jb,\ VA2old?va2; \\
\quad\quad [jb.0 \longrightarrow mp\downarrow \\
\quad\quad [] \underline{else} \longrightarrow CCODE2?(eq, sg); \\
\quad\quad\quad mp := \neg va2 \ \wedge (jb.1 \wedge \neg eq \vee jb.2 \wedge eq \vee jb.3 \wedge sg \vee \\
\quad\quad\quad\quad\quad jb.4 \wedge (sg \vee eq) \vee jb.5 \wedge \neg sg \wedge \neg eq \vee jb.6 \wedge \neg sg) \\
\quad\quad];\ MP!mp, \\
\quad\quad [\neg mp \longrightarrow \overline{EX} \longrightarrow EX,\ e\uparrow \ | \neg \overline{EX} \longrightarrow e\downarrow] [mp \longrightarrow skip]; \\
\quad\quad E!e \\
\quad\quad]
\end{array}
$$

We **factor out** the *va* and *va2* computation from *FETCH3* into process VA. But now, since both the original process and the factored out process use *mp*, *va* and *e* we have to split the channels that send these values (MPE for *mp* and *e* and VA for *va*). Channel MP of process MPE gets split into channels MP0 and MP1 on the receiving end. Process VA computes the value of *va*. It

also sends a copy of this bit on channel VAold. This bit is used by both the original process and the factored out process. Thus, channel VAold is split on the receiving end into channels VAinternal0 and VAinternal1. If *mp=true*, in process VA we can set *va* and *va2* to false, since they either are false to start with or they do not matter.

$$FETCH3 \equiv FETCH4 \;||\; VA$$

$$\begin{array}{l}
VA \equiv \;\; VA!false, \;\; VAold!0, \;\; VAold2!\underline{anything}; \\
\quad *[MP1?mp, \;\; VAinternal1?va, \;\; E1?e; \\
\quad\quad TOSS!mp; \\
\quad\quad [\neg mp \;\wedge\; e \;\longrightarrow\; va\uparrow, \;\; va2\uparrow \\
\quad\quad []\neg mp \;\wedge\; \neg e \;\wedge\; \neg va \;\longrightarrow\; va\downarrow, \;\; va2\downarrow \\
\quad\quad []\neg mp \;\wedge\; \neg e \;\wedge\; va \;\longrightarrow\; va\downarrow, \;\; va2\uparrow \\
\quad\quad []mp \;\longrightarrow\; va\downarrow, \;\; va2\downarrow \\
\quad\quad]; \\
\quad\quad VA!va, \;\; VAold!va, \;\; VAold2!va2 \\
\quad]
\end{array}$$

$$\begin{array}{l}
FETCH4 \equiv PC!\underline{init_pc}; \\
\quad\quad *[MP0?mp, \;\; VAinternal0?va, \;\; E0?e; \\
\quad\quad\quad [\neg mp \longrightarrow F?ins, G?bl, FB?sign, pc2 := pc + 4 \\
\quad\quad\quad []mp \longrightarrow skip \\
\quad\quad\quad]; \\
\quad\quad\quad [\neg mp \wedge ins.i \longrightarrow target := (pc_{31..28}, 4{*}(IMHI?, JIMM?)), bd\uparrow, bj := \underline{no_branch} \\
\quad\quad\quad []\neg mp \wedge ins.r \longrightarrow CCODE1!c3.jr, CCODE3?target, bd\uparrow, bj := \underline{no_branch} \\
\quad\quad\quad []\neg mp \wedge ins.b \longrightarrow BRIMM?brimm, bd\uparrow, \\
\quad\quad\quad\quad [sign \longrightarrow bj := \underline{taken_branch}[]\neg sign \longrightarrow bj := \underline{not_taken_branch}] \\
\quad\quad\quad\quad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1[]bl.1 \longrightarrow CCODE1!c3.branch2]; \\
\quad\quad\quad\quad target := pc + 4{*}brimm \\
\quad\quad\quad []\neg mp \wedge ins.e \longrightarrow bd\downarrow, bj := \underline{no_branch} \\
\quad\quad\quad []mp \longrightarrow bd\downarrow, bj := \underline{mispredict} \\
\quad\quad\quad],
\end{array}$$

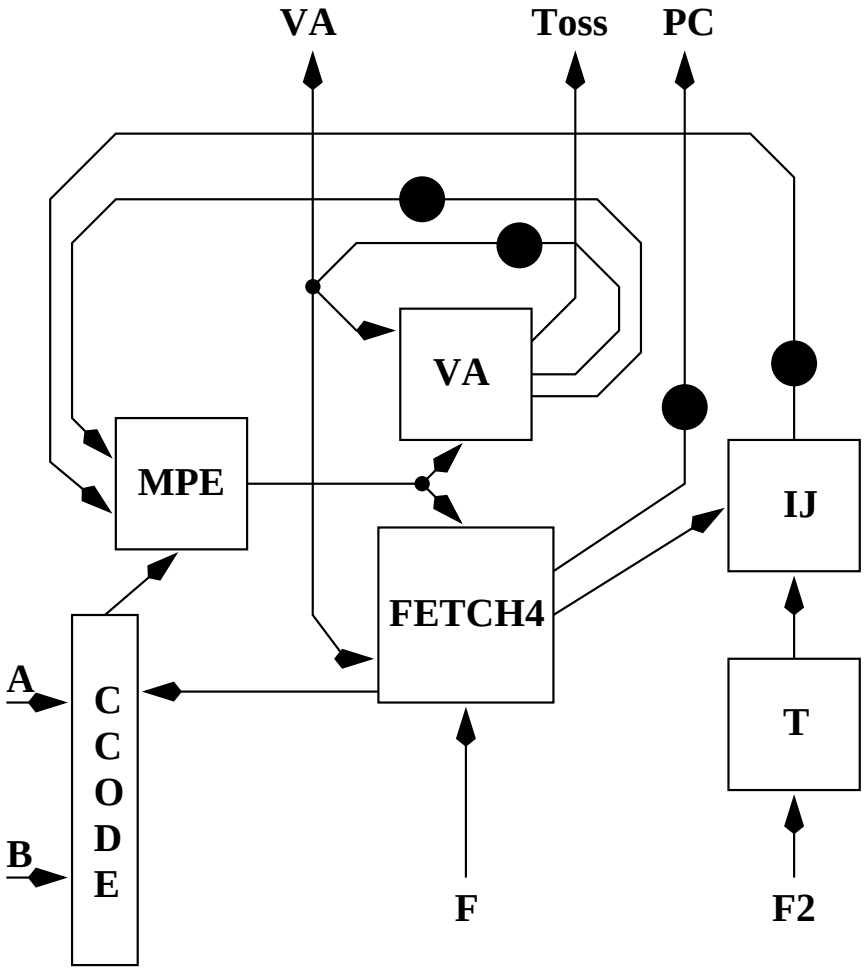


Figure 2: *Partial decomposition of the FETCH*

$$
\begin{array}{l}
[\neg mp \ \wedge \ bl.2 \longrightarrow \ C!pc2 \ [] \ \underline{else} \ \longrightarrow \ skip]; \\
[\neg mp \wedge e \longrightarrow pc := ex\_pc - 4 \\
[]\neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2 \\
[]\neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target \\
[]mp \longrightarrow pc := nottaken \\
]; \\
PC!pc, \ BD!bd, \ BB!bj \\
]
\end{array}
$$

Note that because on a mispredict we do not use *pc2*, we can increment it every cycle (i.e., we need no control for it). We **factor out** the *target* and *pc2* computation . We change each assignment to *target* and *pc2* to a communica-

tion. The branch target is not always read in; thus, we need to send control information to the process that generates it.

$$FETCH4 \equiv FETCH5 \;\;||\;\; PCINC \;\;||\;\; TARGETB$$

$$PCINC \equiv *[PCINC1?pc;\;\; PCINC2!(pc+4)]$$

$$\begin{aligned}
TARGETB \equiv *[&PCI4?c4,\;\; TARGETB1?pc;\\
&[c4.no \;\longrightarrow\; skip\\
&[]c4.yes \;\longrightarrow\; BRIMM?brimm,\;\; TARGETB2!(pc+4*brimm)\\
&]\\
]&
\end{aligned}$$

$$\begin{aligned}
FETCH5 \equiv\; & PC!\underline{init_pc},\;\; PCINC1!\underline{init_pc},\;\; TARGETB1!\underline{init_pc};\\
& *[MP0?mp,\;\; VAinternal0?va,\;\; E0?e,\;\; PCINC2?pc2;\\
& [\neg mp \longrightarrow F?ins, G?bl, FB?sign[]mp \longrightarrow skip];\\
& [\neg mp \wedge ins.i \longrightarrow C4!c4.no, target := (pc2_{31..28}, 4*(IMHI?, JIMM?)),\\
& \qquad bd\uparrow, bj := \underline{no_branch}\\
& []\neg mp \wedge ins.r \longrightarrow C4!c4.no, CCODE1!c3.jr, CCODE3?target,\\
& \qquad bd\uparrow, bj := \underline{no_branch}\\
& []\neg mp \wedge ins.b \longrightarrow C4!c4.yes, bd\uparrow,\\
& \quad [sign \longrightarrow bj := \underline{taken_branch}[]\neg sign \longrightarrow bj := \underline{not_taken_branch}]\\
& \quad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1[]bl.1 \longrightarrow CCODE1!c3.branch2];\\
& \quad TARGETB2?target\\
& []\neg mp \wedge ins.e \longrightarrow C4!c4.no, bd\downarrow, bj := \underline{no_branch}\\
& []mp \longrightarrow C4!c4.no, bd\downarrow, bj := \underline{mispredict}\\
&],\\
& [\neg mp \;\wedge\; bl.2 \longrightarrow\; C!pc2\;[]\;\underline{else}\;\longrightarrow\; skip];\\
& [\neg mp \wedge e \longrightarrow pc := ex_pc - 4\\
& []\neg mp \wedge \neg e \wedge j \wedge \neg va \longrightarrow pc := target, nottaken := pc2\\
& []\neg mp \wedge \neg e \wedge (\neg j \vee va) \longrightarrow pc := pc2, nottaken := target\\
& []mp \longrightarrow pc := nottaken\\
&];\\
& PC!pc,\;\; PCINC1!pc,\;\; TARGETB1!pc,\;\; BD!bd,\;\; BB!bj\\
&]
\end{aligned}$$

Note that by doing the previous transformation we introduced concurrency: while the main process decodes the instruction, the PCINC process increments the program counter and the TARGETB process computes the branch target (in case of a branch).

We **project** on variables *pc*, *target*, *pc2* and *nottaken* separating the datapath from the rest of the control. If as the result of the projection, there are selection statements that are made undeterministic, we need to introduce extra control information to make them deterministic (since the initial process did not have undeterminism). For this reason, we introduced channels C1, C2 and LINK between process PCM and BJ.

$$FETCH5 \equiv PCM \;||\; BJ$$

$$\begin{aligned}
PCM \equiv\; & PC!\underline{init_pc},\;\; PCINC1!\underline{init_pc},\;\; TARGETB1!\underline{init_pc};\\
& *[C1?c1, C2?c2, LINK?link, PCINC2?pc2;\\
& \quad [c1.jump \;\longrightarrow target := (pc2_{31..28}, 4*(IMHI?, JIMM?))\\
& \quad []c1.jr \;\longrightarrow CCODE3?target\\
& \quad []c1.branch \;\longrightarrow TARGETB2?target\\
& \quad []c1.else \;\longrightarrow skip\\
& \quad],\\
& \quad [link.1 \;\longrightarrow\; C!pc2\;\; []link.0 \;\longrightarrow\; skip];\\
& \quad [c2.exception \;\longrightarrow pc := ex_pc - 4\\
& \quad []c2.jnotva \;\longrightarrow pc := target, nottaken := pc2\\
& \quad []c2.notjva \;\longrightarrow pc := pc2, nottaken := target\\
& \quad []c2.mispred \;\longrightarrow pc := nottaken\\
& \quad];\\
& \quad PC!pc,\;\; PCINC1!pc,\;\; TARGETB1!pc\\
&]
\end{aligned}$$

$$
\begin{array}{l}
BJ \equiv *[MP0?mp,\ \ VAinternal0?va,\ \ E0?e; \\
\quad [\neg mp \ \longrightarrow\ F?ins,\ G?bl,\ FB?sign \ [] mp \ \longrightarrow\ skip]; \\
\quad [\neg mp \ \wedge\ ins.i \ \longrightarrow\ C1!c1.jump, C4!c4.no, BD!true, BB!\underline{no\_branch} \\
\quad [] \neg mp \ \wedge\ ins.r \ \longrightarrow\ C1!c1.jr, C4!c4.no, CCODE1!c3.jr, BD!true, BB!\underline{no\_branch} \\
\quad [] \neg mp \ \wedge\ ins.b \ \longrightarrow\ C1!c1.branch, C4!c4.yes, BD!true, \\
\qquad [sign \longrightarrow bj := \underline{taken\_branch} [] \neg sign \longrightarrow BB!\underline{not\_taken\_branch}] \\
\qquad [bl.0 \vee bl.2 \longrightarrow CCODE1!c3.branch1 [] bl.1 \longrightarrow CCODE1!c3.branch2]; \\
\quad [] \neg mp \ \wedge\ ins.n \ \longrightarrow\ C1!c1.else, C4!false, BD!false, BB!\underline{no\_branch} \\
\quad [] mp \ \longrightarrow\ C1!c1.else, C4!c4.no, BD!false, BB!\underline{mispredict} \\
\quad], \\
\quad [\neg mp \ \wedge\ bl.2 \ \longrightarrow\ LINK!link.1 [] \underline{else} \ \longrightarrow\ LINK!link.0], \\
\quad [\neg mp \ \wedge\ e \ \longrightarrow\ C2!c2.exception \\
\quad [] \neg mp \ \wedge\ \neg e \ \wedge\ j \ \wedge\ \neg va \ \longrightarrow\ C2!c2.jnotva \\
\quad [] \neg mp \ \wedge\ \neg e \ \wedge\ (\neg j \ \vee\ va) \ \longrightarrow\ C2!c2.notjva \\
\quad [] mp \ \longrightarrow\ C2!c2.mispred \\
\quad] \\
]
\end{array}
$$

We do one more transformation to process PCM which eliminates the temporary variable *target*. *c2.jnotva* and *c1.else* are exclusive since if *c2.jnotva* is true then j is true in the same cycle, this implies that the instruction is different from a *nop*; thus *c1.else=false*.

Variable *nottaken* is only used on a branch; for this reason, we save it only on a branch (this optimization is very important because now *nottaken* is not written in the most common case).

$$
\begin{aligned}
PCM \equiv\ & PC!\underline{init\_pc},\ \ PCINC1!\underline{init\_pc},\ \ TARGETB1!\underline{init\_pc}; \\
& *[C1?c1,\ C2?c2,\ LINK?link,\ PCINC2?pc2; \\
& \quad [c2.notjva \longrightarrow pc := pc2 \\
& \quad []\ c2.jnotva \wedge c1.jump \longrightarrow pc := (pc2_{31..28}, 4*(IMHI?, JIMM?)) \\
& \quad []\ c2.jnotva \wedge c1.jr \longrightarrow CCODE3?pc \\
& \quad []\ c2.jnotva \wedge c1.branch \longrightarrow TARGETB2?pc \\
& \quad []\ c2.mispred \longrightarrow pc := nottaken \\
& \quad []\ c2.exception \longrightarrow pc := ex\_pc - 4], \\
& \quad [link.1 \longrightarrow C!pc2\ []\ link.0 \longrightarrow skip], \\
& \quad [c2.jnotva \wedge c1.branch \longrightarrow nottaken := pc2 \\
& \quad []\ c2.notjva \wedge c1.branch \longrightarrow nottaken := target]; \\
& \quad PC!pc,\ \ PCINC1!pc,\ \ TARGETB1!pc \\
& \ ]
\end{aligned}
$$

After all these steps of decomposition we broke down the initial CHP into several individual processes:

$$
\begin{aligned}
FETCH \equiv\ & CCODE\ ||\ MPE\ ||\ IJ\ ||\ T\ ||\ VA\ ||\ PCINC\ || \\
& TARGETB\ ||\ PCM\ ||\ BJ
\end{aligned}
$$

Most of these processes have the template we described earlier; thus we can implement them directly. Some of them still need extra decomposition. There are only 3 stages on the forward latency of the FETCH on the most common case (BJ - copy trees - PCM). Given the high complexity of the unit, this is a very pleasing result.

1.6 Further decomposition and production rules

One of the challenges of implementing the FETCH was the double constraint of very low forward latency (since it is part of the critical *fetch loop*) and low cycle time (since it is used every cycle). In general, if forward latency is not an issue, a slow (in terms of throughput) pipeline stage could be further decomposed in several simpler pipeline stages which could be made faster. However, that was not an option with the FETCH. The decomposition process was done keeping these issues in mind.

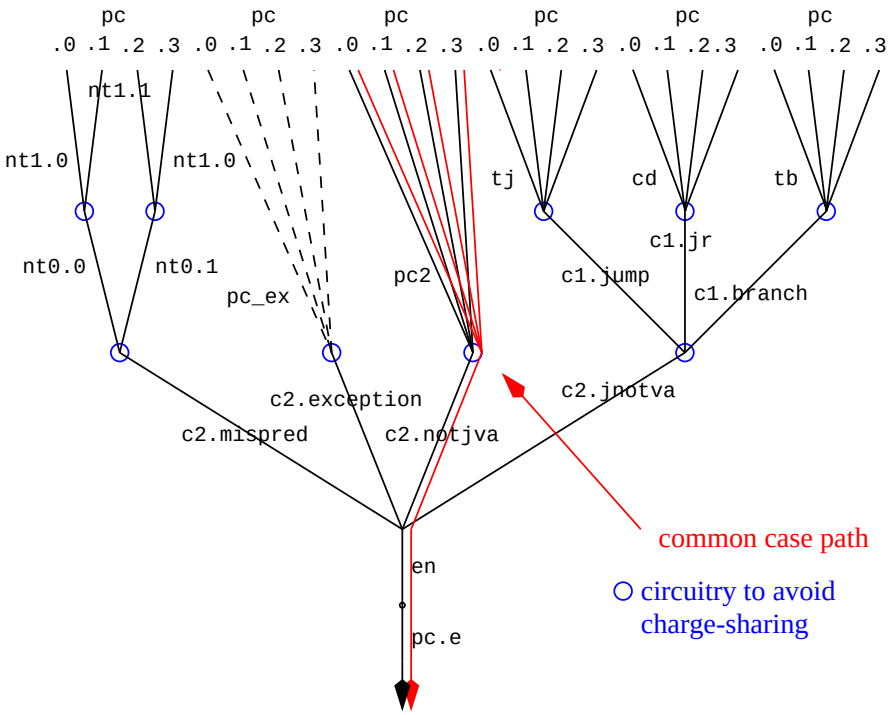


Figure 3: *Minimal-gate network for the program counter*

1.6.1 PCM process

The most difficult process to implement with a high throughput and low latency was the PCM. This unit was highly optimized for the general case (non branch) both at production-rule level (reshuffling) and at transistor level (sizing) since they constitute more than 4/5 of the instructions executed by this unit. We used the *hybrid-buffer* reshuffling (described in detail in the next chapter): full-buffering on common case (9 transitions set, 7 transitions reset) and halfbuffering on the other cases (9 transitions set, 9 transitions reset). We used quad-rail logic to implement data communications and dual-rail flip-flops to implement the *nottaken* register. The datapath of the FETCH in general, the PCM unit in particular, is also pipelined *horizontally* (as described in [3]), synchronized in chunks of 8 bits. The minimal-gate network for the inverted program counter is shown in figure 4. The production rules for two bits (encoded in a 1-of-4 code) of the inverted program counter are:

$$\neg en \wedge \neg pc.e \longrightarrow \_pc.0\uparrow$$
$$\neg en \wedge \neg pc.e \longrightarrow \_pc.1\uparrow$$
$$\neg en \wedge \neg pc.e \longrightarrow \_pc.2\uparrow$$
$$\neg en \wedge \neg pc.e \longrightarrow \_pc.3\uparrow$$

$$\begin{aligned}
&en \wedge pc.e \wedge \\
&\quad (c2.notjva \wedge pc2.0 \vee \\
&\qquad c2.jnotva \wedge (c1.jump \wedge tj.0 \vee c1.jr \wedge cd.0 \vee c1.branch \wedge tb.0) \vee \\
&\qquad c2.mispred \wedge nottaken1.0 \wedge nottaken0.0 \vee \\
&\qquad c2.exception \wedge exe[0]) \longrightarrow _pc.0\downarrow \\
&en \wedge pc.e \wedge \\
&\quad (c2.notjva \wedge pc2.1 \vee \\
&\qquad c2.jnotva \wedge (c1.jump \wedge tj.1 \vee c1.jr \wedge cd.1 \vee c1.branch \wedge tb.1) \vee \\
&\qquad c2.mispred \wedge nottaken1.0 \wedge nottaken0.1 \vee \\
&\qquad c2.exception \wedge exe[1]) \longrightarrow _pc.1\downarrow
\end{aligned}$$

$$\begin{aligned}
&en \wedge pc.e \wedge \\
&\quad (c2.notjva \wedge pc2.2 \vee \\
&\qquad c2.jnotva \wedge (c1.jump \wedge tj.2 \vee c1.jr \wedge cd.2 \vee c1.branch \wedge tb.2) \vee \\
&\qquad c2.mispred \wedge nottaken1.1 \wedge nottaken0.0 \vee \\
&\qquad c2.exception \wedge exe[2]) \longrightarrow _pc.2\downarrow \\
&en \wedge pc.e \wedge \\
&\quad (c2.notjva \wedge pc2.3 \vee \\
&\qquad c2.jnotva \wedge (c1.jump \wedge tj.3 \vee c1.jr \wedge cd.3 \vee c1.branch \wedge tb.3) \vee \\
&\qquad c2.mispred \wedge nottaken1.1 \wedge nottaken1.1 \vee \\
&\qquad c2.exception \wedge exe[3]) \longrightarrow _pc.3\downarrow
\end{aligned}$$

The array exe[0..3] is used to set the exception handler's address. Only one of exe[0], exe[1], exe[2] or exe[3] is true for a given quad.

In terms of transistor sizing, the common case path was sized significantly bigger that the other paths. In principle, each path was sized loosely proportional with the frequency of occurrences of that path in normal execution. Note that the doted line in the minimal-gate network means that at the time of instantiation of each quad, only one of the lines is wired to the output, based on the particular value of that quad in the exception program counter.

Special care was taken to eliminate charge-sharing bumps from output nodes. Almost all of the internal nodes of the minimal gate network are precharged by a series of 2 p-transistors connected on one side to the internal node to be precharged and on the other side to Vdd. The inputs to the 2 p-transistors are *en* and *pc.e.*

Despite the high complexity of this process, in our implementation we successfully met both the latency and throughput targets. Hspice measurements show that the forward latency of the PCM is 0.3641ns, and its throughput is about 315MHz, both numbers are for the most common case of execution (in 0.6μ CMOS).

1.6.2 CCODE process

Another interesting implementation is that of the CCODE which requires a 32-bit comparator with an auxiliary bypass channel.

Since the comparator is used only on branches or jump-registers, its throughput is not critical; however, its latency is. The CCODE, when used to compute the branch condition, produces 2 bits of output. One, the sign bit, is very easy to compute: it is a copy of the most significant bit of operand A. However, the other bit, the equality bit is more difficult to compute with low forward latency, because we are limited by the number of transistors in the pull-down chain.

In general, in a standard stage we compute the inverted outputs with N-transistors; then, we pass them through inverters to get the positive sense of the output (and also to amplify the signal). However, in case of the comparator we decided to do some logic with P-transistors, too; substituting the inverters of the output with more complicated logic. This allowed us to compare bundles of 4 bits in the first stage of the CCODE. The quad-rail inputs are a0, a1, b0, b1 and the dual-rail output is r. The signal c3.1 is used to compare against zero, while c3.2 to compare the two input channels. The production rules for the data logic of the first stage for 4 bits are:

$$\neg go \longrightarrow \_r0.1\uparrow$$
$$\neg go \longrightarrow \_r0.0\uparrow$$

$$\begin{aligned}
&go\wedge\\
&\quad(c3.1 \wedge a0.0\vee\\
&\qquad c3.2 \wedge (a0.0 \wedge b0.0 \vee a0.1 \wedge b0.1 \vee a0.2 \wedge b0.2 \vee a0.3 \wedge b0.3)) \longrightarrow _r0.1\downarrow
\end{aligned}$$

$$\begin{aligned}
&go\wedge\\
&\quad(c3.1 \wedge (a0.1 \vee a0.2 \vee a0.3)\vee\\
&\qquad c3.2 \wedge (a0.0 \wedge (b0.1 \vee b0.2 \vee b0.3)\vee\\
&\qquad\qquad a0.1 \wedge (b0.0 \vee b0.2 \vee b0.3)\vee\\
&\qquad\qquad a0.2 \wedge (b0.0 \vee b0.1 \vee b0.3)\vee\\
&\qquad\qquad a0.3 \wedge (b0.0 \vee b0.1 \vee b0.2))) \longrightarrow _r0.0\downarrow
\end{aligned}$$

$$\neg go \longrightarrow \_r1.1\uparrow$$
$$\neg go \longrightarrow \_r1.0\uparrow$$

$$\begin{aligned}
&go\wedge\\
&\quad(c3.1 \wedge a1.0\vee\\
&\qquad c3.2 \wedge (a1.0 \wedge b1.0 \vee a1.1 \wedge b1.1 \vee a1.2 \wedge b1.2 \vee a1.3 \wedge b1.3)) \longrightarrow _r1.1\downarrow
\end{aligned}$$

$$\begin{aligned}
&go\wedge\\
&\quad(c3.1 \wedge (a1.1 \vee a1.2 \vee a1.3)\vee\\
&\qquad c3.2 \wedge (a1.0 \wedge (b1.1 \vee b1.2 \vee b1.3)\vee\\
&\qquad\qquad a1.1 \wedge (b1.0 \vee b1.2 \vee b1.3)\vee\\
&\qquad\qquad a1.2 \wedge (b1.0 \vee b1.1 \vee b1.3)\vee\\
&\qquad\qquad a1.3 \wedge (b1.0 \vee b1.1 \vee b1.2))) \longrightarrow _r1.0\downarrow
\end{aligned}$$

$$\begin{aligned}
&\neg_r0.1 \ \wedge\ \neg_r1.1 \longrightarrow r.1\uparrow\\
&\ _r0.1 \ \vee\ \ _r1.1 \longrightarrow r.1\downarrow
\end{aligned}$$

$$\begin{aligned}
&\neg_r0.0 \ \vee\ \neg_r1.0 \longrightarrow r.0\uparrow\\
&\ _r0.0 \ \wedge\ _r1.0 \longrightarrow r.0\downarrow
\end{aligned}$$

When implementing these production-rules, we precharged some of the internal nodes of the minimal-gate network to avoid charge-sharing. It should be noted that computing in p transistors is not only slower but also less noise

resistant than computing in N-transistors. Also, because the inverters are eliminated, the slew rates of the output signals are worse.

The second stage of the comparator combines the 8 bits generated by the 8 bundles of 4 bits in the first stage into one bit, true if the two 32-bit channels are equal and false if they are not. This stage is similar in implementation with the first stage.

Hspice measurement show that the forward latency of the 32-bit comparator is 1.044ns for the true case (all input bits equal) and 0.985ns for the false case. The data dependent forward-latency is due to the length difference of the pull-down networks for the true and false case. A similar comparator (on 20-bits) used in the cache tag comparison of the Caltech MiniMIPS is described in [4].

1.6.3 MPE process

An interesting issue in implementing this unit was the isolation of the synchronizer on the EX channel, and its implementation using and arbiter. We further decompose MPE into

$$MPE \equiv MPL \;||\; EXE$$

$$\begin{aligned}
MPL \equiv *[&JB?jb,\ VA2old?va2;\\
&[jb.0 \longrightarrow mp\downarrow\\
&[]\underline{else} \longrightarrow CCODE2?(eq, sg);\\
&\quad mp := \neg va2 \wedge (jb.1 \wedge \neg eq \vee jb.2 \wedge eq \vee jb.3 \wedge sg \vee\\
&\qquad jb.4 \wedge (sg \vee eq) \vee jb.5 \wedge \neg sg \wedge \neg eq \vee jb.6 \wedge \neg sg)\\
&];\ MP!mp,\\
&[\neg mp \longrightarrow R?e\ []\ mp \longrightarrow e\downarrow];\\
&E!e\\
&]
\end{aligned}$$

$$\begin{aligned}
EXE \equiv *[[&[\overline{R} \wedge \overline{EX} \longrightarrow R!true;\ EX\\
&|\overline{R} \wedge \neg\overline{EX} \longrightarrow R!false\\
&]]
\end{aligned}$$

Process MPE is reshuffled in such a way that we read in the R channel independently of mp; however, we acknowledge it only if mispredict is false,

thus avoiding to lose and exception on a true mispredict while simplifying the implementation of variable e. This transformation is correct because, on a mispredict, the exception bit is not used in the logic of the next stages (it gets only completed and acknowledged).

The EXE process was designed by Rajit Manohar [3]. The hand-shaking expansion for it is:

$$
\begin{aligned}
EXE \equiv *[[& EX.i \longrightarrow [R.e];\ R.t\uparrow;\ EX.e\uparrow;\ [\neg EX.i \wedge \neg R.e];\ R.t\downarrow;\ EX.e\downarrow \\
& | R.e \longrightarrow R.f\uparrow;\ [\neg R.e];\ R.f\downarrow \\
&]]
\end{aligned}
$$

where channel EX is ($EX.i$,$EX.e$) and channel R is ($R.f$,$R.t$,$R.e$). Now we introduce the variables u and v, such that we can factor out the arbiter

$$
\begin{aligned}
EXE \equiv *[[& EX.i \longrightarrow u\uparrow; [R.e]; R.t\uparrow; EX.e\uparrow; [\neg EX.i]; u\downarrow; [\neg R.e]; R.t\downarrow; EX.e\downarrow \\
& | R.e \longrightarrow v\uparrow; R.f\uparrow; [\neg R.e]; v\downarrow; R.f\downarrow \\
&]]
\end{aligned}
$$

We decompose EXE into the standard (as described in [1]) arbiter with inputs $EX.i$ and $R.e$

$$
\begin{aligned}
ARB \equiv *[[& EX.i \longrightarrow u\uparrow;\ [\neg EX.i];\ u\downarrow \\
& | R.e \longrightarrow v\uparrow;\ [\neg R.e];\ v\downarrow \\
&]]
\end{aligned}
$$

and

$$
\begin{aligned}
MPX \equiv *[[& u \longrightarrow [R.e];\ R.t\uparrow;\ [\neg R.e];\ EX.e\uparrow;\ [\neg u];\ R.t\downarrow;\ EX.e\downarrow \\
& | v \longrightarrow R.f\uparrow;\ [\neg v];\ R.f\downarrow \\
&]]
\end{aligned}
$$

$$EXE \equiv MPX \ || \ ARB$$

The complete bubble-reshuffled PRs for process MPX could be find in [3], while the PRs for ARB could be found in [1].

1.7 Slack matching

Since the FETCH executes four main, quite different instructions, thee slack-matching of the datapath is different for each of these instructions, since

the paths data takes on these instructions is different. However, the slack-matching of the control is not dependent on the type of instruction being executed, since each token follows the same path inside the control. It turned out that we can optimally slack-match the datapath only for no branch instructions (luckily, that is the most common case) and jump immediates.

The slack-matching for the control is simple, by adding 5 half-buffers on channels VAold and VAold2 and 6 half-buffers on channel JB, the fetch control loop is optimally slack-matched (having one token for eight half-buffer stages as suggested in [6]).

For the datapath, in the most common case, the critical loop is PCM–2 half-buffers–one-full-buffer–INC–one full-buffer back to PCM. This loop has one token and an equivalent of eight half-buffer stages; thus it is optimally slack-matched. For a jump immediate, there is no slack-mismatch, since the immediate comes in on a private channel, directly into the PCM. In case of a jump register, the operand on A comes in 8 stages later, while the control for it arrives 3 stages later. These split-merge paths could be slack-matched only by adding 5 stages of half-buffering (or about 2 full-buffers) on the channel controlling CCODE, since there is no way to reduce the 8 stage latency for the register file. However, we chose not to add those buffers, since even if we would slack match that split-merge path, we would not be able to match it with the BJ–copy trees–PCM path, on which we cannot add buffering (since it is on the critical forward path).

In case of a branch, even though the inputs to the branch adder are early, its control is not; thus, it takes 4 stages to generate the outputs that go into the PCM. For this reason, the split-merge paths BJ–copy trees–PCM and BJ–4 stages of data–PCM is mismatched with three stages. There is no way to make the BJ–4 stages of data–PCM path shorter, since the number of stages for the adder is fixed. On the other hand, the path BJ–copy trees–PCM cannot be made longer since it is on the critical *fetch loop*. This three stage mismatch, match up in the TOSS with the 3 stage mismatch due to the register file latency. So, in principle, the two tokens to be merged arrive at the same time, both 3 stages late. Thus, just by reducing the mismatch inside the fetch would not improve performance on branches. Note that even if the paths BJ–copy trees–PCM and BJ–4 stages of data–PCM are mismatched, depending on the

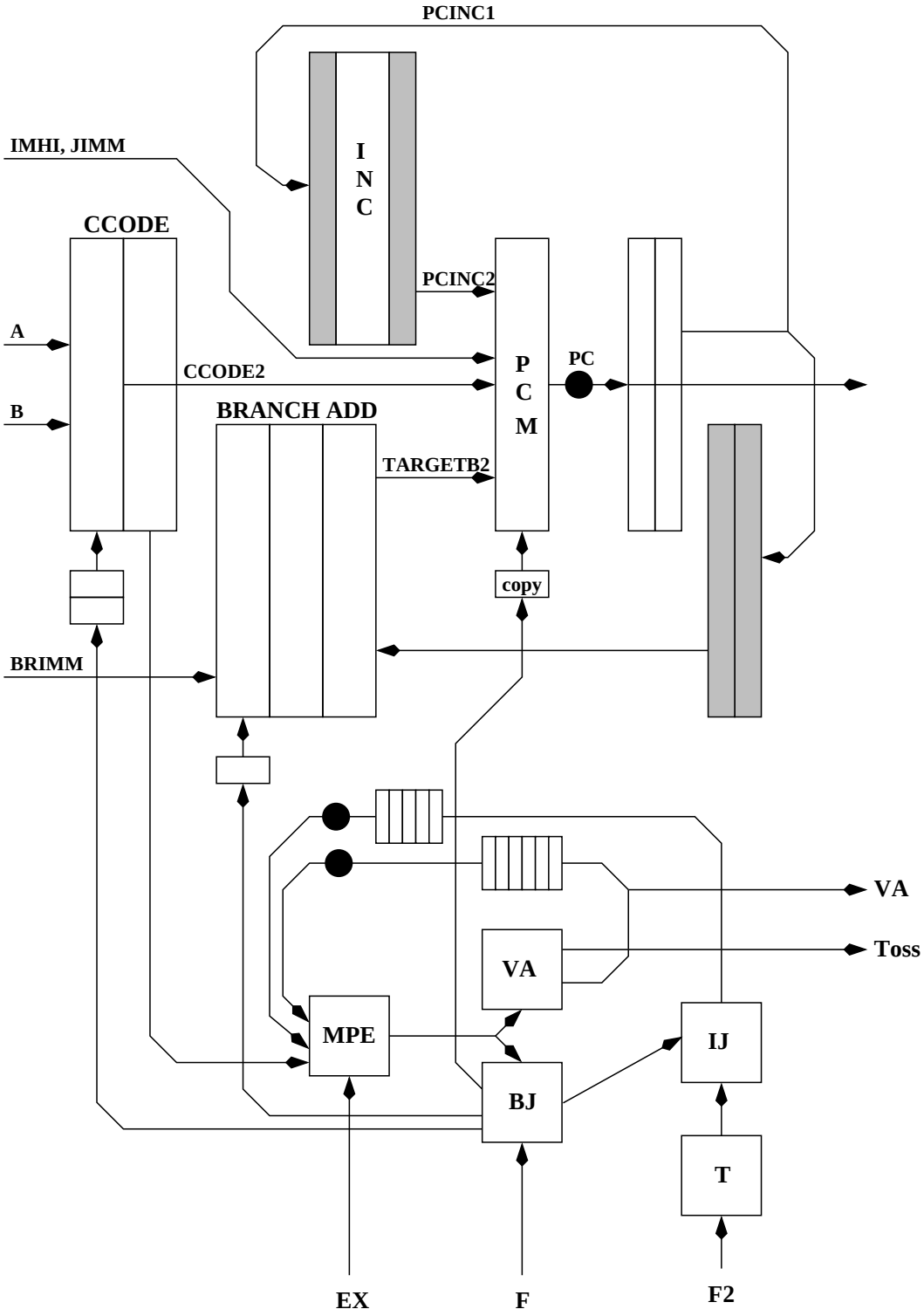


Figure 4: *Slack-matching of the FETCH*

prediction mechanism, sometimes the incremented *old* program counter is sent out as new program counter. In this case, the 3 stage latency mismatch does not get exposed on the forward latency of the program counter; however, in order to complete up the hand-shake, the (late) inputs from the branch adder do need to be read in (and saved in the *nottaken* register) making the cycle on a branch longer (i.e. postponing the beginning of the next cycle).

1.8 Summary and conclusions

In this chapter we showed how we implemented, based on a high level CHP description the high performance asynchronous FETCH of the Caltech MiniMIPS. By describing the step-by-step decomposition we showed how we introduce pipelining and concurrency to a sequential CHP program. We pointed out special challenges encountered along the design process: double-sided constrains in terms of latency and throughput, selective slack-matching, avoiding charge sharing.

Hspice measurements show that (in 0.6μ CMOS) the throughput of the FETCH for common case is about 313MHz, while the forward latency is 1.0337ns (where 0.4032ns are due to BJ, 0.2664ns are due to the copy between the control and datapath and 0.3641ns are due PCM); thus both constrains on throughput and latency were met.

2 Charge sharing in LR-buffers

2.1 Introduction

In this chapter we will present the most common charge sharing cases that we were faced with in the design of the Caltech MiniMIPS microprocessor. We will present some of the solutions we have developed to solve these problems.

2.2 Charge sharing in LR-buffers

Charge sharing is the parasitic effect of charge redistribution among internal and output nodes of an operator. Given the relative strength of N-transistors over P-transistors, most of the computation in a high performance CMOS circuit is done using N-transistors. Thus, the N-transistor networks tend to get more complicated and there are comparatively more internal nodes that could affect the output nodes. In the Caltech MiniMIPS microprocessor most of the circuits are aggressively sized to achieve low forward latency and gates are heavily shared to reduce transistor loading. These measures are needed to achieve high performance. However, the down side of transistor sizing is that internal nodes tend to have significant capacitance compared to the capacitance of the output nodes. This can cause serious charge sharing problems on the output nodes. If an incorrect transition would propagate, the circuits would very likely deadlock.

In this report we will focus our attention to N-type charge sharing, i.e. charge sharing occurring in N-transistor networks. P-type charge sharing is less common and most of the techniques presented for the N-type charge sharing could be used to prevent P-type charge sharing, as well. The most important N-Type charge sharing occurs in the pull-down network of the precharge logic. Based on the time of appearence, these charge sharing events can be divided into four types: *N-dn set phase charge sharing*, *N-up set phase charge sharing*, *N-dn reset phase charge sharing* and *N-up reset phase charge sharing*. Most of the charge sharing problems in the Caltech MiniMIPS fall within these four categories.

To explain the problem more concretely let us consider the following example:

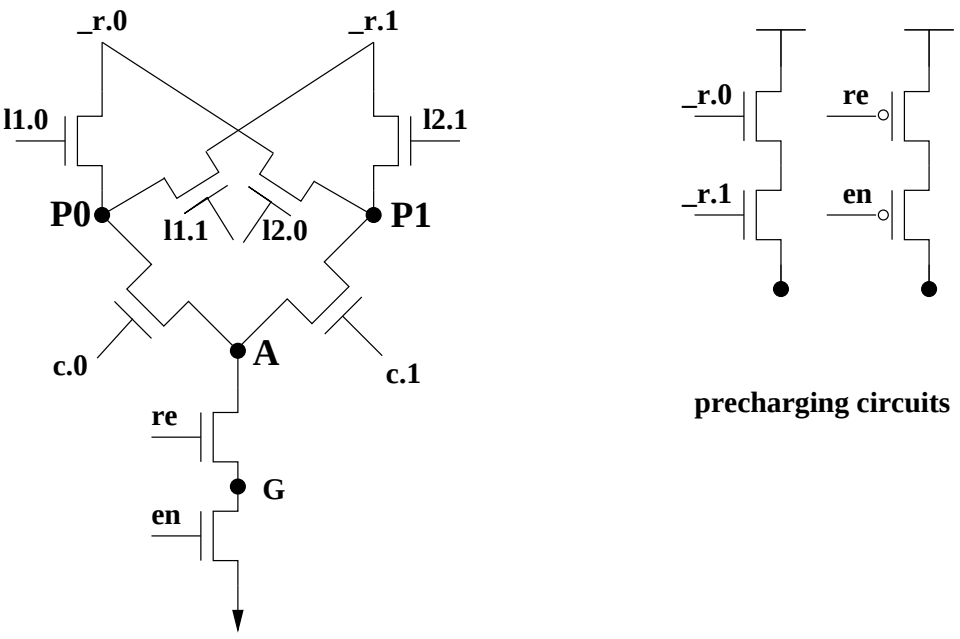


Figure 5: *Minimal gate network for Example1*

$$Example1 \equiv *[C?c;\ \ [c=0 \ \longrightarrow\ R!(L1?)[]c=1 \ \longrightarrow\ R!(L2?)]]$$

The output on channel R could be generated with the PRs (followed by inverters):

$$\begin{aligned} en \ \wedge\ re \ \wedge\ (c.0 \ \wedge\ l1.0 \ \vee\ c.1 \ \wedge\ l2.1) &\longrightarrow\ \_r.0\downarrow \\ en \ \wedge\ re \ \wedge\ (c.0 \ \wedge\ l1.1 \ \vee\ c.1 \ \wedge\ l2.0) &\longrightarrow\ \_r.1\downarrow \\ \neg en \ \wedge\ \neg re &\longrightarrow\ \_r.0\uparrow \\ \neg en \ \wedge\ \neg re &\longrightarrow\ \_r.1\uparrow \end{aligned}$$

implemented with the minimal gate network of Figure 5.

For the first two scenarios to be described, consider *Example1* implemented using half-buffer reshuffling. The corresponding HSE is ($n()$ neutrality of data, $v()$ validity of data):

$$\begin{aligned} *[[&(c.0 \wedge L1 \vee c.1 \wedge L2) \wedge Re]; R\Uparrow; \\ &([c.0 \wedge v(L1)]; Ce\downarrow, L1e\downarrow), ([c.1 \wedge v(L2)]; Ce\downarrow, L2e\downarrow); \\ &\qquad [\neg Re]; R\Downarrow; \\ &([n(C) \wedge n(L1)]; Ce\uparrow, L1e\uparrow), ([n(C) \wedge n(L2)]; Ce\uparrow, L2e\uparrow)] \end{aligned}$$

Internal nodes P0, P1, A and G might have significant capacitance compared to _r.0 and _r.1. Indeed, if L1 and L2 are 1of4 channels, nodes P0 and P1 would have even more capacitance. Assume P0, A and G are low $\{\neg P0 \wedge \neg A \wedge \neg G\}$ (they were discharged in the previous cycles), _r.0 and _r.1 are high $\{\_r.0 \wedge \_r.1\}$ (they were precharged) and that the next tokens to arrive will cause $l1.0\uparrow$ and $c.0\uparrow$. If $l1.0\uparrow$, $c.0\uparrow$ and $re\uparrow$ happen earlier than $en\uparrow$, i.e. there exists a state $\{l1.0 \wedge c.0 \wedge re \wedge \neg en\}$, the charge from node P0, A and G will be shared with node $\_r.0$. Since node $\_r.0$ was high, this charge sharing causes the potential on node $\_r.0$ to drop. If the drop reaches the logic threshold of the gate connected to it, an early firing will occur and the false transition will propagate. We categorize this type of charge sharing as *N-dn set-phase charge sharing.*

Next, consider the following scenario. P1 is high from the previous cycle, and en, re, inputs $l1.0$ and $c.0$ become high. This causes one of the production rules for $\_r.0\downarrow$ to fire and $\_r.0$, P0, A and G are discharged to GND $\{\neg \_r.0 \wedge \neg P0 \wedge P1 \wedge \neg A \wedge \neg G\}$. After some transitions, the pull-down of $\_r.0$ shuts off $(\{\neg en \vee \neg re \vee \neg c.0 \vee \neg l1.0\})$ and a 0-token arrives on channel L2 $\{l2.0\}$. This is possible in normal operation since channels L1 and L2 are not synchronized. Under these circumstances the charge from node P1 is dumped on $\_r.0$ and the potential of node $\_r.0$ increases; possibly causing an early firing. We categorize this type of charge sharing as *N-up set-phase charge sharing.*

For the next two scenarios consider *Example1* implemented using full-buffer reshuffling. The corresponding HSE is:

$$
\begin{aligned}
*[[(c.0 \wedge L1 \vee c.1 \wedge L2) \wedge Re]; R\Uparrow;\\
&([c.0 \wedge v(L1)]; Ce\downarrow, L1e\downarrow), ([c.1 \wedge v(L2)]; Ce\downarrow, L2e\downarrow);\\
&\qquad ([\neg Re]; R\Downarrow),\\
&([n(C) \wedge n(L1)]; Ce\uparrow, L1e\uparrow), ([n(C) \wedge n(L2)]; Ce\uparrow, L2e\uparrow)]
\end{aligned}
$$

The main difference between these cases and the previous ones is that a new token can arrive on any of the input channels before the output has reset. This is due to the fact that $l1.e\uparrow$ or $l2.e\uparrow$ do not wait for $\{\neg R\}$. Consider the following scenario. $\_r.0$, P0, A and G are discharged in the set phase of the cycle and then the inputs are acknowledged in both senses before the output channel R resets, i.e. while still $\{\neg \_r.0 \wedge \_r.1\}$. This allows a new input token to arrive on channel *L1* and *C*. If $l1.1\uparrow$ and $c.0\uparrow$ the charge on nodes P0,

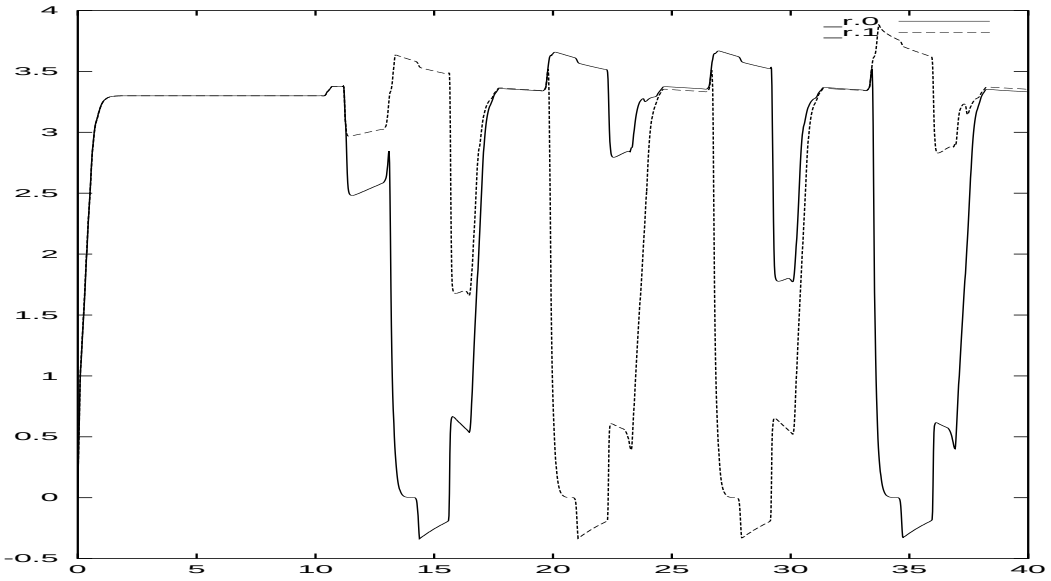


Figure 6: *Charge sharing in full-buffers*

A and $\_r.1$ is shared and the potential on node $\_r.1$ drops, possibly causing a misfiring. We categorize this type of charge sharing as *N-dn reset-phase charge sharing* (see Figure 6).

For the last type of charge sharing consider the following scenario. The production rule for $\_r.0\downarrow$ fires and $\_r.0$, P0, A and G are discharged to GND $\{\neg\_r.0 \wedge \neg P0 \wedge \neg A \wedge \neg G\}$. The input channels are acknowledged in both senses before the output channel R resets (so far, same as the previous scenario). Now, a new token on channels C and L2 could arrive. If $c.1\uparrow$ and $l2.0\uparrow$ the charge on nodes $\_r.0$, P1 and A is shared, causing the potential on node $\_r.0$ to increase (due to P1), potentially early-firing (see Figure 6). We categorize this type of charge sharing as *N-up reset-phase charge sharing.*

The *N-dn set-phase charge sharing* and *N-up set-phase charge sharing* are common to all types of buffer reshufflings, while the *N-dn reset-phase charge sharing* and *N-up reset-phase charge sharing* are common to *hybrid-buffer* (see next chapter) and *full-buffer* reshufflings.

In general, there are two main strategies to deal with charge sharing. The first is to make the ratio of output capacitance over internal capacitance bigger. This strategy works well for all four types of charge sharing. The second strategy is to make the potential difference between the output node and the

internal nodes smaller. This strategy works for *N-dn set phase charge sharing* and, as it will be presented in this report, it could be made to work for *N-dn reset phase charge sharing* as well. In general, it does not help *N-up charge sharing*. Furthermore, in some cases fixing *N-dn charge sharing* could worsen the *N-up charge sharing*. Most of the time, to solve a charge sharing problem on a given operator, both strategies can be applied at the same time.

Making the ratio of output capacitance to worst-case internal capacitance bigger can be done in two ways. We can increase the output capacitance, or we can decrease the worst-case internal capacitance by shrinking and folding the internal N diffusion or by reordering the signals in parts of the operator. When applied to an already sized circuit, both of these operations might affect the performance of the circuit by slowing down the output transition of the given N transistor network.

Reducing the potential difference between internal and output nodes could be done by precharging the internal nodes to the value of the output. One way this could be applied to the *N dn set-phase charge sharing* is to precharge the internal nodes in the reset-phase of the cycle. This technique was first introduced in our circuits by Mika Nyström. There are several types of circuits that could be used to precharge the internal nodes, using either P or N transistors. Figure 5 shows two different ways in which nodes P0, P1 and A could be precharged (these nodes might need to be staticized, too). By precharging the internal nodes in the reset-phase of the cycle there potential gets close to Vdd. When the new inputs arrive in the set phase of the cycle and expose the internal nodes, there is no significant potential difference between the output and the internal nodes; thus, there is no significant charge sharing. The closer the potential of the internal nodes to the output node, the smaller the charge sharing will be. If the precharging is done using P-type transistors, the potential of the internal nodes could be brought to Vdd. If N-type transistors are used, the potential of the internal nodes could be brought to Vdd-$N_{threshold}$. This method works well for the *N dn set-phase charge sharing* and was widely used in several circuits in the Caltech MiniMIPS processor.

The precharging technique presented before relies on the fact that by the time new inputs arrive at the input of the stage the internal nodes were already precharged. More precisely, the state allowing $R \Downarrow$ (resetting the output and

turning the precharge circuits on) should happen before $l.e\uparrow$ (allowing a new token on the input). This condition is satisfied in the *half-buffer* reshuffling; however, it is not satisfied in the *hybrid buffer* or the *full buffer* reshufflings. In the later cases the previous technique is completely ineffective, since the precharge circuits might not turn on in time.

Assume we precharge the internal nodes with a charge Q (for this argument it is not relevant the way we precharge the nodes). If Q is close to Vdd the *N-dn charge sharing* is attenuated; however, the *N-up charge sharing* is worsened. Consequently, if Q is close to GND, the *N-up charge sharing* is attenuated and the *N-dn charge sharing* becomes worse. The sum of the *N dn charge sharing* and *N up charge sharing* is proportional with charge Q. By picking the right Q, we can influence the ratio between the *N-dn* and *N-up* charge sharings. In particular, we can make one less severe at the expense of the other. This solution is not practical because getting the right Q on the internal nodes is difficult and the solution does not reduce the sum of the two charge sharings. Furthermore, keeping an internal node with charge Q is not feasible. Under normal operating conditions it is possible that the nodes precharged to Q get exposed to the output (without a firing) and drift to Vdd-$N_{threshold}$ or get exposed to GND directly and are discharged. Next, we will present a feasible solution.

2.3 Solving the *N dn charge sharing* problem for full-buffering

The hand-shaking expansion for a full-buffer is:

$$*[[f(L) \wedge Re]; \; R\Uparrow; \; [v(L)]; \; Le\downarrow; \; ([\neg Re]; \; R\Downarrow), \; ([n(L)]; \; Le\uparrow)]$$

We could syntactically add to any full-buffering circuit a "fake" half-buffering input channel and transform it into a hybrid-buffering one. The *hybrid-buffer* (see next chapter) has the property that if the non-fullbuffering channel is closest to the output nodes and the internal nodes are properly precharged no *N-dn charge sharing* occurs. This "fake" channel would be added to the minimal gate network of the initial circuit in such a way that it is the closest channel to the output nodes. By doing this and precharging the internal nodes as presented before, both *N-dn charge sharing* types could be eliminated from

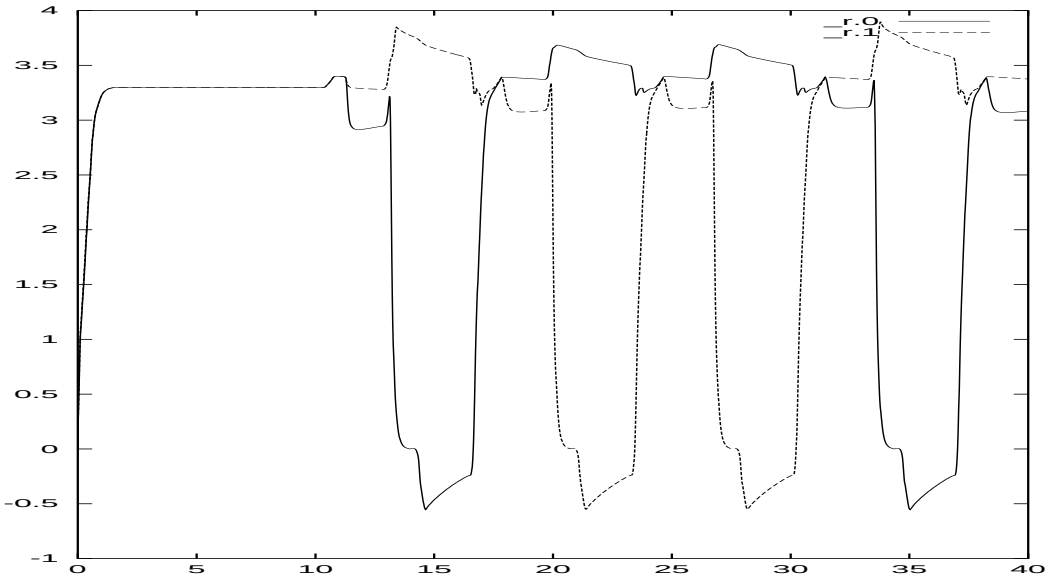


Figure 7: *No more N dn charge sharing in hybrid and full buffers*

any full-buffering circuit. The "fake" input channel could be as simple as a one-rail channel connected to a bitgenerator. The hand-shake on this channel could be always made fast enough not to be on the critical path. In this way, while having all the advantages of full-buffering, the *N-dn charge sharing* problems have been eliminated. This solution comes for the cost of an added series transistor in the pull-down network (the transistor corresponding to the "fake" channel).

Taking the idea of a "fake" channel further, we notice that we can achieve the same results just by adding a state variable to the hand-shaking expansion. The state variable can be used to gate the internal nodes until the proper precharging is finished. We add the state variable _Rv to the initial hand-shaking expansion:

$$*[[\_Rv \wedge en \wedge f(L) \wedge Re]; \;\; R\Uparrow; \;\; \_Rv\downarrow; \;\; [v(L)]; \;\; Le\downarrow; \;\; [\neg\_Rv]; \;\; en\downarrow$$
$$([\neg Re]; \;\; R\Downarrow; \;\; \_Rv\uparrow), \;\; ([n(L)]; \;\; Le\uparrow); \;\; en\uparrow]$$

The signal $\_Rv$ will be part of the minimal-gate-network of output R. It will be the closest signal to this output. Now, when $\_Rv\uparrow$ the internal nodes are exposed to the output. If the precharge circuit turns on when $\{\neg Re \wedge \neg en\}$ or $R\Downarrow$ then, by the time $\_Rv\uparrow$ the internal nodes were properly precharged

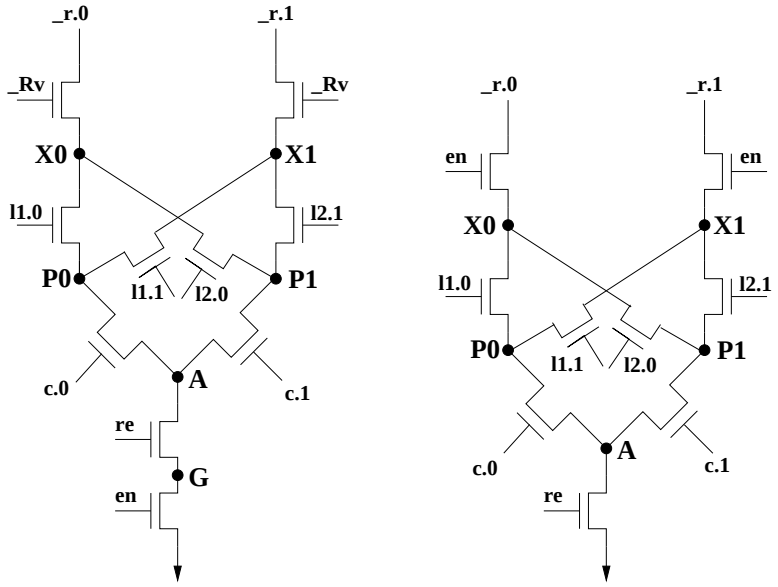


Figure 8: *Minimal gate network for Example1 without N dn charge sharing*

and there will be no *N-dn charge sharing*. The disadvantage of this solution is that it still has an extra N-transistor in series in the pull-down network. The result of adding variable *_Rv* to the original circuit is shown in Figure 7.

Another state variable that could be used instead of *_Rv* is *en*. This choice was suggested by Andrew Lines and Mika Nyström. This solution eliminates the extra gate needed in the minimal gate network and uses *en* to gate internal nodes from outputs. However, initially *en* was shared at the base of the minimal gate network. Now, the transistor for *en* needs to be duplicated for each output rail. This adds load on signal *en* and slows down the internal cycle of the unit. However, in general internal cycles could be made non-critical and the use of *en* as a gating signal becomes feasible.

Figure 8 shows the minimal-gate-network of *Example1* implemented with the *full-buffer* reshuffling for both choices of the state variables.

Both choices of state variables as gating signal work well in practice. The choice between them is circuit dependent and might vary from case to case.

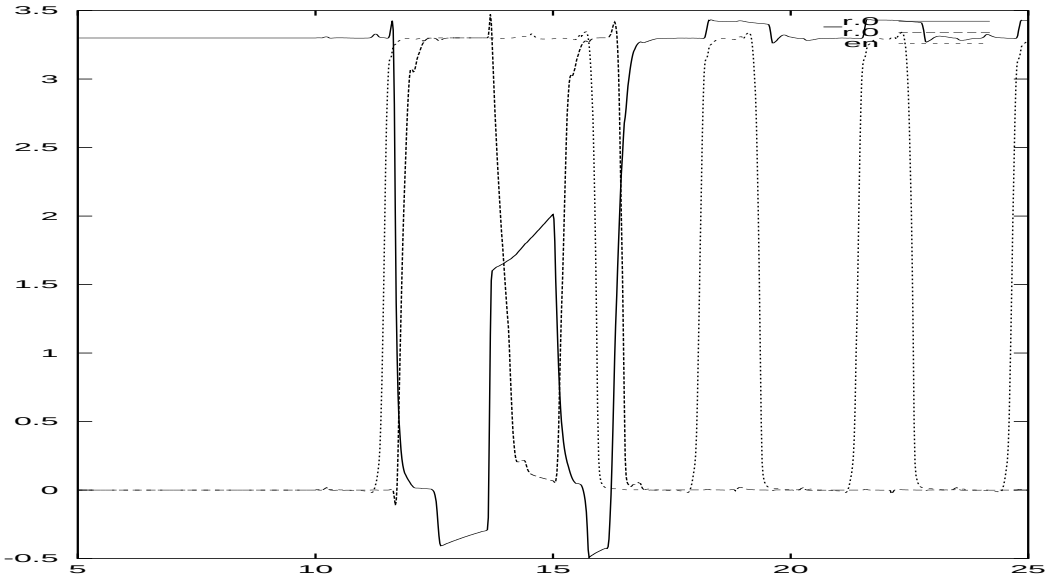


Figure 9: *Circuit failing due to N-up charge sharing*

2.4 *N-up charge sharing*, a discussion

N-up charge sharing is common to full-buffering logic and disjunctive half-buffering logic. This type of charge sharing appears less frequently and less severely in our analog circuit simulations. However; this is not to say that it could not lead to circuit failures the same way as *N-dn charge sharing* does. Consider a 3 way merge where the data is encoded using a 1-of-4 code.

$$Example2 \equiv *[C?c; [c = 0 \longrightarrow R!(L1?)[]c = 1 \longrightarrow R!(L2?)[]c = 2 \longrightarrow R!(L3?)]]]$$

We implemented this circuit using *half-buffer* reshuffling using reasonable transistor sizes and we observed it misfunction due to N-up charge sharing. We exposed the worst case N-up charge sharing due to which the output inverter early fired. Figure 9 shows a spice simulation of the failing circuit behavior. Notice that $\_r.0\uparrow$ when en is still true, while $\_r.0$ was supposed to reset after state $\{\neg en \wedge \neg Re\}$.

Given the standard *half-buffer* reshuffling, the *N-up charge sharing* occurs between states $\{\neg \_R\}$ and $\{\neg en \wedge \neg Re\}$. Furthermore, for such an event to affect a given output node there should be an alternative path from GND to that output node. When a given output node is pulled down to GND, all nodes along that path are pulled down to GND as well. Disregarding some

analog effects (as capacitive coupling or substrate leakage) their charge cannot participate in the *N-up charge sharing* event. There should exist some other charged internal nodes (not on the path just discharged) that can dump their charge on the output and possibly make it fire early. The further these internal nodes are from the output and the smaller their charge is, the less effect their charge will have. This is because undriven internal nodes further away from the output influence the output indirectly, through other nodes in the path; thus, part of their initial charge goes on nodes other then the output.

The above description already suggests several ways to reduce *N-up charge sharing*. If there is only one path to a given output node, that node will not suffer *N-up charge sharing*. Even if there are alternative paths to a given output node, those nodes that are common among the different paths cannot contribute to *N-up charge sharing*, since they get discharged every time one of the output rails is pulled down. In particular, for a minimal gate network where *en* and *re* are shared at the bottom of the shared gate network, none of the nodes next to gates *en* or *re* can contribute to *N-up charge sharing* (in the minimal gate network for *Example1* nodes A and G). Furthermore, by reordering the inputs such that *en* or *re* (or both) are closest to the output nodes, the *N-up charge sharing* could be further reduced. The same effect could be achieved by having exclusive signals closest to the output. This would allow only one path to the output to be exercised during a given cycle.

The voltage on an internal node of an N-transistor network (without explicit precharge circuits attached to it) varies within GND and Vdd-$N_{threshold}$ (ignoring coupling). However, if an internal node is precharged with P transistors, not only does its capacitance gets bigger (because the capacitance of the precharge circuit itself) but its charge can also get higher than Vdd-$N_{threshold}$ and close to Vdd. In this case, the charge exposed to an output node in case of a *N-up charge sharing* event would be higher than what it would have been possible if there would have been no precharge circuit attached to the internal node. This shows that by fixing the *N-dn charge sharing* using P-type precharge circuits, the *N-up charge sharing* gets worse. For this reason, if *N-up charge sharing* is possible, P-type precharge circuits should be avoided.

In a pipeline stage with disjunctive logic, it is possible to produce an output before all the inputs arrive. The late inputs could expose charged internal

nodes to the output that just got pulled down, contributing to the *N-up charge sharing*. If the validity of the input channels could be included in the logic, the output is not produced until all inputs arrived; thus, once the output transitioned, there will be no more internal nodes exposed to it. As a result, there will be no *N-up charge sharing*. In conclusion, when using *half-buffer* reshuffling *N-up charge sharing* could be eliminated by including the validity of the inputs in the pull-down network of the output.

If *en* is the closest signal to the output in the shared gate network, the *N-up charge sharing* event happens within the same state interval for a *half buffer* reshuffling as for a *full buffer* reshuffling. The reason is that once $en\downarrow$ the output nodes are disconnected from the pull-down network; thus, the tokens from the next cycle (the ones that can possibly arrive due to the full-buffering property of the handshake) cannot expose any internal nodes on the current output.

If $\neg en \wedge \neg re$ is used as pull-up for the inverted outputs, the charge on the capacitance between the P-transistors *en* and re gets added to the worst case *N-up charge sharing*. This could be the case if the closest P-transistor to the output transitions before the P-transistor closest to Vdd does. This effect could be reduced by folding and reordering the pull-up transistors or completely eliminated by introducing a single P-transistor as pull-up.

2.5 Do combinational operators charge share ?

We would like to find out if we could eliminate charge sharing by transforming non-combinational operators that are susceptible to charge sharing into combinational operators. There are several ways an operator could be made combinational. In particular, a node $\_c$ specified by the production rules $A \rightarrow \_c\downarrow$ and $B \rightarrow \_c\uparrow$ could be transformed into a combinational node specified by the production rules $A|c \wedge \neg B \rightarrow \_c\downarrow$, $B|c \wedge \neg A \rightarrow \_c\uparrow$, $\_c \rightarrow c\downarrow$ and $\neg\_c \rightarrow c\uparrow$.

Consider an LR-buffer implemented using the *full-buffer* handshaking expansion:

$$*[[en \wedge Re \wedge f(L)^1]; \ \_R\downarrow^2; \ R\Uparrow, \ Rv\uparrow; \ [v(L) \wedge Rv]; \quad Le\downarrow; \ en\downarrow;$$
$$([\neg en \wedge \neg Re^3]; \quad \_R\uparrow^4; \ R\Downarrow, \ Rv\downarrow^5), \ ([n(L)]; Le\uparrow^6); \ [\neg Rv]; \ en\uparrow]$$

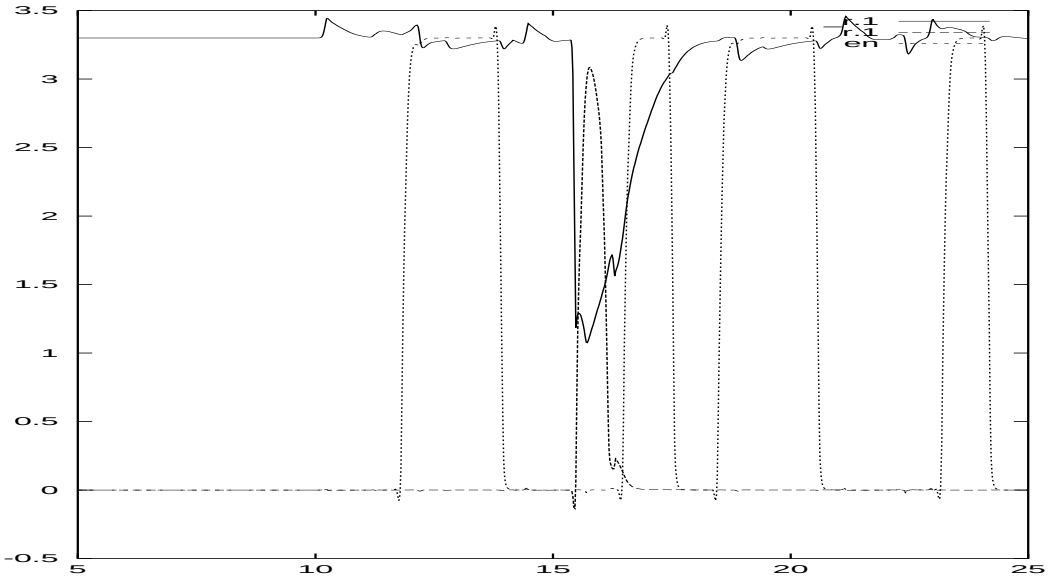


Figure 10: *Combinational node failing due to N-dn charge sharing*

The worse charge sharing events happen in the pull-down network of the inverted output ($\_R$). As we described it before, there are several types of charge sharings that can occur: *N-dn set phase* between states 5 and 1 or 6 and 1 (whichever state interval is larger will include the other one), *N-dn reset phase* between states 2 and 3 and *N-up* between states 2 and 3.

As explained before, the inverted output ($\_R$) could be made fully combinational. We try to avoid this solution for the time being. Instead, we would like to add simple vacuous firings to the handshaking expansion in such a way that when a given charge sharing event occurs, the inverted output $\_R$ is driven. Assume channel R is encoded as a 1 of n channel. By adding $R.i \wedge (en|Re) \rightarrow \_R.i \downarrow$ for each rail, we cover the state space between states 2 and 3 for the rail that was pulled down. By adding $\neg\_R.0|...|\neg\_R.i-1|\neg\_R.i+1|...|\neg\_R.n-1 \rightarrow \_R.i \uparrow$ for each rail we cover the state space between states 2 and 4. By adding $\neg en \wedge \neg Rv|\neg Re \wedge \neg R.i \rightarrow \_R.i \uparrow$ for each rail we cover the state space between states 5 and 1.

If we would add to the inverted output $\_R$ logic the first two vacuous firings described before; then, when *N-dn reset phase* and *N-up* charge sharing occurs, the affected nodes would be driven. Thus, the extra charge resulting from the charge sharing would be driven off through the open path to one of

the power supplies. However, it turns out that the mere fact that a node is driven does not significantly affect the charge sharing at the moment it occurs. It is true that after a period of time, the charge is driven off the affected node; but that doesn't happens instantenuosly. In case of a severe charge sharing event, there might be sufficient time for a misfiring to propagate, before the parasitic charge gets driven off.

We implemented *Example2* using *half-buffer* reshuffling and we made the inverted outputs combinational. We observed this circuit fail due to *N-dn charge sharing* (Figure 10). Note that at the moment of firing $\_r.0\downarrow$, signal *en* is still false; as a result, signal $\_r.0$ fires early.

Further analysis showed that even if there was a minor positive effect in the added vacuous firings, most of it came from the fact that the output capacitance of the operator was increased. On the other hand, vacuous firings do not contribute to the computation, and the added capacitance slows the circuits down. If increasing the output vs inside capacitance is an option, reducing the size of the pull-down network or increasing the output inverter yields better results. In conclusion, adding vacuous firings to nodes susceptible of charge sharing was found inferior to other techniques used to reduce or eliminate charge sharing.

3 Hybrid LR-buffer. Very high slack buffers

3.1 Introduction

In the Caltech MiniMIPS microprocessor the most widely used reshuffling was the *half-buffer* reshuffling ([7]). However, under certain circumstances other buffer reshufflings are more desirable. In this chapter we will present two of these buffers. The first type, the *hybrid-buffer* has slack one on all but one channel and when the proper gate ordering is picked, it has an outstanding robustness to charge-sharing. The second type, the *standard* and *double buffer* have a static slack of $1\frac{1}{2}$ and 2 respectively, and a high dynamic slack.

3.2 Hybrid LR-buffer

Consider the following CHP:

$LR - buffer \equiv *[L1?, L2?; R!]$

The *half-buffer* reshuffling corresponding to this CHP is:

$$half\_buffer \equiv *[[f(L1 \wedge L2) \wedge Re]; \; R\Uparrow; \; [v(L1) \wedge v(L2)]; \; L1e\downarrow, L2e\downarrow; \\ [\neg Re]; \; R\Downarrow; \; [n(L1) \wedge n(L2)]; \; L1e\uparrow, L2e\uparrow]$$

The *hybrid buffer* reshuffling for the same CHP is:

$$hybrid\_buffer \equiv *[[f(L1 \wedge L2) \wedge Re]; \; R\Uparrow; \; [v(L1) \wedge v(L2)]; \; L1e\downarrow, L2e\downarrow; \\ ([\neg Re]; \; R\Downarrow; \; [n(L1)]; \; L1e\uparrow), ([n(L2)]; \; L2e\uparrow)]$$

This later hand-shaking expansion has the property that $L2e\uparrow$ does not wait for $\neg Re$. The processing of the current token can finish early and the hand-shake on the new token can start sooner. This behavior changes the slack properties of the buffer as seen from channel *L2*. A new token is allowed to be present on channel *L2*; while the hand-shake from the previous cycle is not finished on the output channel R. The static slack of channel *L2* is increased from $\frac{1}{2}$ (*half buffer*) to 1.

Avoiding deadlock in a system or increasing dynamic slack on some channels might necessitate increasing the static slack on some channels (since the bigger the static slack, the bigger the dynamic slack). However, by adding buffers the forward latency on the corresponding paths is increased; an undesired side-effect. The *hybrid-buffer* reshuffling offers a solution to increase

static slack on some channels without affecting the forward latency. Furthermore, it does it for no area penalty, as opposed to a *full-buffer* where the same effect would be achieved with a significant area cost.

The *hybrid-buffer* was successfully used in several places in the Caltech MiniMIPS processor including: the PCM unit of the Fetch, several places in the Fetch control, the HiLo register of the MultDiv and the LCache process of the ICACHE.

3.3 Implementation of the *hybrid buffer*

Consider the previous LR-buffer with two input channels L1 and L2 and one output channel R implemented using *half buffer* reshuffling with a global enable (*en*). The left enables $L1e$ and $L2e$ could be implemented with the following PRs:

$$\begin{aligned} L1v &\wedge Rv &&\longrightarrow L1e\downarrow \\ \neg L1v &\wedge \neg Rv &&\longrightarrow L1e\uparrow \end{aligned}$$

$$\begin{aligned} L2v &\wedge Rv &&\longrightarrow L2e\downarrow \\ \neg L2v &\wedge \neg Rv &&\longrightarrow L2e\uparrow \end{aligned}$$

where $L1v$, $L2v$ are the validity signals of input channel $L1$ respectively $L2$ and Rv is the validity of output channel R. This implementation could be easily transformed into a *hybrid buffer* with channel $L2$ full-buffering by changing the PRs of $L2e$ into:

$$\begin{aligned} L2v &\wedge en \wedge Rv &&\longrightarrow L2e\downarrow \\ \neg L2v &\wedge \neg en &&\longrightarrow L2e\uparrow \end{aligned}$$

while keeping the PRs of $L1e$ unchanged. Using this type of transformation all but one channel of a half-buffer stage could be made full-buffering with practically no area cost.

3.4 Solving the *N-dn reset-phase charge sharing* problem for hybrid-buffering

The full-buffering channels of a *hybrid buffer* allow new tokens to arrive before the outputs corresponding to the current tokens have reset. This can

cause the same type of charge sharing problems present in the full-buffer. The *N dn set phase* charge sharing could be solve as presented in the previous chapter by precharging highly capacitive internal nodes. The basic idea in solving the *N dn reset-phase charge sharing* is to notice that if the internal nodes are exposed to the outputs only after they have been precharged, there will be no *N dn charge sharing*.

In the case of the *hybrid buffer* reshuffling, channel *L1* is half-buffering; thus, when a new token arrives on this channel the state $R \Downarrow$ was reached already. If the internal nodes are exposed only after a token on channel *L1* arrives (and not after a token on channel *L2* arrives) the internal nodes could be precharged properly the same way as for the *N dn set-phase charge sharing*. As a result, if in the minimal gate network, we make the outer channel to be the one that is half-buffering, the precharging methods presented before will work properly.

In conclusion, for the general case of *hybrid buffer* reshuffling (when there are several input channels; all but one full-buffering) by reordering the gates in such a way that the half-buffering channel is the closest to the outputs and by precharging the internal nodes (as shown before) both types of *N dn charge sharing* could be eliminated.

3.5 High slack buffers

For the rest of this chapter we present a discussion on two high slack buffers and their application to the design of asynchronous VLSI systems. High slack buffers are mostly used as slack matching buffers. We will present two different LR-buffer reshufflings, one with static slack $\frac{3}{2}$ and dynamic slack from 0.4 to 1.0 and the other with static slack 2 and dynamic slack from 0.4 to 1.25.

Most of the stages of a high-speed asynchronous VLSI system are desirable to have low forward latency and low dynamic slack. This property allows several stages of the pipeline to work on the same token at full throughput. However, there are cases in which the opposite property is desirable: high latency and high dynamic slack. In particular, high dynamic slack is needed for slack matching and implementing state feedback loops (which are a particular case of slack-matching). In case of slack-matching, the bigger the mismatch the more slack-matching buffers are needed; however, adding the buffers increases

both area and energy dissipation. Thus, we would like to have high dynamic slack buffers of reasonable size (i.e. with minimal area per dynamic slack).

We reason about dynamic slack using the throughput versus number of tokens graph. A detailed discussion about these graphs and slack matching in general could be found in [7]. One property of interest is that for a given pipeline the shape of the graph is a triangle or a trapezoid (resulting from a triangle with its top cut off). The base of the triangle (or trapezoid) is given by the static slack of the pipeline, while the top of the figure is given by the maximum throughput of the pipeline. The static slack of an entire pipeline depends on the static slack of the individual components it is made out of. The maximum throughput is inversely proportional to the critical path within the pipeline. The critical path depends both on the internal structure of each component (internal cycle and backward latency) and the spacing of the tokens in the pipeline.

3.6 Standard and double buffers

There are at least two ways to increase the dynamic slack of an LR-buffer: first, by reducing the number of transitions per cycle (increases the throughput and raises the top of the triangle (or trapezoid)) and second, by increasing the static slack of the buffer (elongates the base of the triangle (or trapezoid)). The number of transitions in a bare buffer is already low. Further reducing the number of transitions would deteriorate the electrical properties of the circuit (especially when eliminating the inverters on the outputs and left acknowledge). In the buffers considered in this report the number of transitions per cycle is kept low, while the static slack of the buffer is increased. The result of this operation is to increase the dynamic slack, the property we are looking for.

The first observation we make is that static slack could be increased only in *quantums*.

Theorem (Quantification of static slack)

Given an LR-buffer operating with a four-phase handshake, its static slack is a multiple of $\frac{1}{2}$.

Given an LR-buffer operating with a two-phase handshake, its static slack is a multiple of 1.

Proof. In an LR-buffer the left acknowledge of L (le) and the right acknowledge of R (re) are synchronized. If they were not synchronized there could be an unbounded number of L actions before an R action. That would imply an unbounded number of different place holders for the values received on channel L. Obviously, this cannot be the case in a finite VLSI circuit.

If le↑ is synchronized with re↓ the slack could $\frac{1}{2}$, $\frac{3}{2}$, ... for the four-phase handshake and 1, 2, ... for the four-phase handshake. If le↑ is synchronized with re↑ the slack could be a positive integer.

A *half buffer* has static slack of $\frac{1}{2}$, while a *full buffer* has a static slack of 1, if its neighbors are other full buffers. We will develop two buffers with static slack $\frac{3}{2}$ and 2, respectively. Given the LR-buffer *[L?;R!] the first reshuffling we chose is:

$$standard\_buffer \equiv *[([l];\ le\downarrow),\ ([\neg re];\ r\downarrow);\ ([\neg l];\ le\uparrow),\ ([re];\ r\uparrow)]$$

We will refer to this reshuffling as the *standard buffer* reshuffling. The *static slack* of this reshuffling is $\frac{3}{2}$.

To analyze the dynamic slack of the *standard buffer* we first need an efficient implementation of this handshaking expansion (in order to keep the top of the triangle high). The PR set we developed is presented below and has 12 transitions per cycle in case the state bit changes and 10 transitions otherwise.

$$\begin{aligned}
&_en \ \wedge\ l.0 \ \longrightarrow\ x.1\downarrow\\
&_en \ \wedge\ l.1 \ \longrightarrow\ x.0\downarrow\\
&\neg x.0 \ \wedge\ (\neg l.0 \ \vee\ \neg_en) \ \longrightarrow\ x.1\uparrow\\
&\neg x.1 \ \wedge\ (\neg l.1 \ \vee\ \neg_en) \ \longrightarrow\ x.0\uparrow\\
&l.0 \ \wedge\ x.0 \ \vee\ l.1 \ \wedge\ x.1 \ \longrightarrow\ _xl\downarrow\\
&\neg Reset \ \wedge\ \neg en \ \wedge\ \neg_xl \ \longrightarrow\ _le\uparrow\\
&_le \ \longrightarrow\ l.e\downarrow
\end{aligned}$$

$$\neg en \ \wedge \ \neg r.e \ \longrightarrow \ \_r.0\uparrow$$
$$\neg en \ \wedge \ \neg r.e \ \longrightarrow \ \_r.1\uparrow$$
$$\_r.0 \ \wedge \ \_r.1 \ \longrightarrow \ xr\downarrow$$
$$\_r.0 \ \longrightarrow \ r.0\downarrow$$
$$\_r.1 \ \longrightarrow \ r.1\downarrow$$
$$\neg xr \ \longrightarrow \ latchout\uparrow$$
$$Reset\_ \ \wedge \ latchout \ \wedge \ \_le \ \longrightarrow \ \_en\downarrow$$
$$\neg \_en \ \longrightarrow \ en\uparrow$$

$$\neg l.0 \ \wedge \ \neg l.1 \ \longrightarrow \ \_xl\uparrow$$
$$Reset \ \vee \ en \ \wedge \ \_xl \ \longrightarrow \ \_le\downarrow$$
$$\neg \_le \ \longrightarrow \ l.e\uparrow$$

$$en \ \wedge \ latchout \ \wedge \ r.e \ \wedge \ x.0 \ \longrightarrow \ \_r.0\downarrow$$
$$en \ \wedge \ latchout \ \wedge \ r.e \ \wedge \ x.1 \ \longrightarrow \ \_r.1\downarrow$$
$$\neg \_r.0 \ \vee \ \neg \_r.1 \ \longrightarrow \ xr\uparrow$$
$$\neg \_r.0 \ \longrightarrow \ r.0\uparrow$$
$$\neg \_r.1 \ \longrightarrow \ r.1\uparrow$$
$$xr \ \longrightarrow \ latchout\downarrow$$
$$\neg Reset\_ \ \vee \ \neg latchout \ \wedge \ \neg \_le \ \longrightarrow \ \_en\uparrow$$
$$\_en \ \longrightarrow \ en\downarrow$$

Using the throughput-vs-tokens graph we determined the peak performance to be achieved at $\frac{2}{3}$ tokens per stage with a dynamic slack of 0.4 to 1.0 (see figure 11).

We can further increase the static slack of the *standard buffer* by observing that the left acknowledge could be reset without waiting for the outputs to reset, the wait could be postponed into the *en* completion tree. The handshaking expansion will be transformed into:

$$double\_buffer \equiv *[([l]; \ le\downarrow), \ ([\neg re]; \ r\downarrow)); \ [re]; \ r\uparrow] \ || \ *[[\neg le]; \ [\neg l]; \ le\uparrow]$$

We will refer to the above reshuffling as the *double buffer* reshuffling. This reshuffling allows a static slack of 2 and for our 12 transition implementation a dynamic slack from 0.4 to 1.25. The peak performance, as in the case of the *standard buffer*, is achieved at $\frac{2}{3}$ tokens per stage (see figure 12).

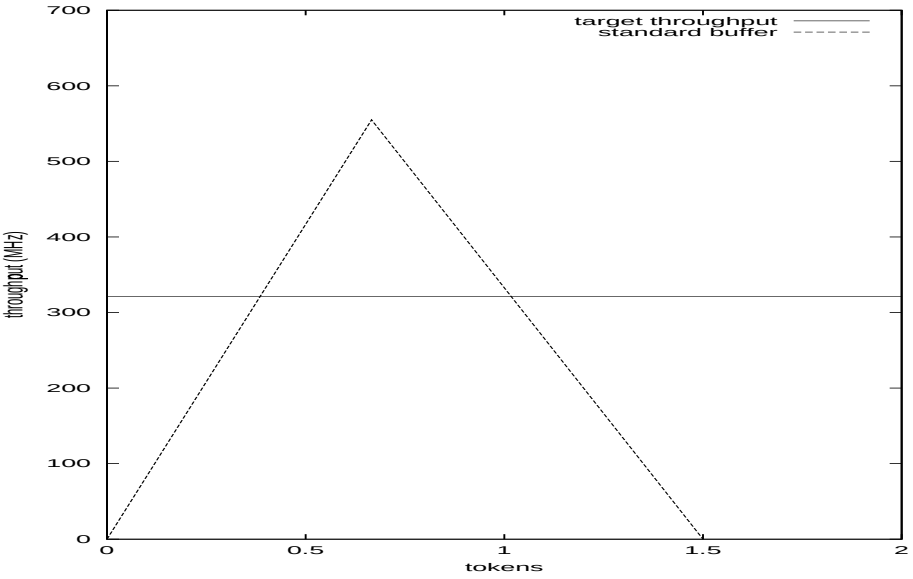


Figure 11: *Dynamic slack in standard buffers*

$$Reset \ \vee \ \_en \ \wedge \ l.0 \ \longrightarrow \ x.1\downarrow$$
$$Reset\_ \ \wedge \ \_en \ \wedge \ l.1 \ \longrightarrow \ x.0\downarrow$$
$$\neg x.0 \ \wedge \ (\neg l.0 \ \vee \ \neg\_en) \ \longrightarrow \ x.1\uparrow$$
$$\neg Reset\_ \ \vee \ \neg x.1 \ \wedge \ (\neg l.1 \ \vee \ \neg\_en) \ \longrightarrow \ x.0\uparrow$$

$$l.0 \ \wedge \ x.0 \ \vee \ l.1 \ \wedge \ x.1 \ \longrightarrow \ \_xl\downarrow$$
$$\neg\_xl \ \wedge \ \neg en \ \wedge \ \neg en\_\_ \ \longrightarrow \ \_le\uparrow$$
$$\_le \ \longrightarrow \ l.e\downarrow$$

$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg r.e \ \longrightarrow \ \_r.0\uparrow$$
$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg r.e \ \longrightarrow \ \_r.1\uparrow$$
$$\_r.0 \ \wedge \ \_r.1 \ \longrightarrow \ xr\downarrow$$
$$\_r.0 \ \longrightarrow \ r.0\downarrow$$
$$\_r.1 \ \longrightarrow \ r.1\downarrow$$
$$\neg xr \ \longrightarrow \ latchout\uparrow$$
$$latchout \ \longrightarrow \ \_latchout\downarrow$$

$$
\begin{aligned}
&\_le \ \wedge \ enn \ \longrightarrow \ \_en\downarrow \\
&\neg\_en \ \longrightarrow \ en\_\_\uparrow \\
&\neg\_latchout \ \wedge \ \neg\_en \ \longrightarrow \ en\uparrow \\
&en \ \longrightarrow \ enn\downarrow \\
\\
&\neg l.0 \ \wedge \ \neg l.1 \ \longrightarrow \ \_xl\uparrow \\
&Reset \ \vee \ en\_\_ \ \wedge \ \_xl \ \longrightarrow \ \_le\downarrow \\
&\neg\_le \ \longrightarrow \ l.e\uparrow \\
\\
&en \ \wedge \ latchout \ \wedge \ r.e \ \wedge \ x.0 \ \longrightarrow \ \_r.0\downarrow \\
&en \ \wedge \ latchout \ \wedge \ r.e \ \wedge \ x.1 \ \longrightarrow \ \_r.1\downarrow \\
&\neg\_r.0 \ \vee \ \neg\_r.1 \ \longrightarrow \ xr\uparrow \\
&\neg\_r.0 \ \longrightarrow \ r.0\uparrow \\
&\neg\_r.1 \ \longrightarrow \ r.1\uparrow \\
&xr \ \longrightarrow \ latchout\downarrow \\
&\neg latchout \ \longrightarrow \ \_latchout\uparrow \\
\\
&\neg Reset\_ \ \vee \ \neg latchout \ \wedge \ \neg\_le \ \wedge \ \neg enn \ \longrightarrow \ \_en\uparrow \\
&\_en \ \longrightarrow \ en\_\_\downarrow \\
&Reset \ \vee \ \_latchout \ \wedge \ \_en \ \longrightarrow \ en\downarrow \\
&\neg en \ \longrightarrow \ enn\uparrow
\end{aligned}
$$

3.7 Standard and double buffers for 1-of-3 and 1-of-4 codes

Both high slack buffers presented before rely for their implementation on a dual-rail state bit. In order to implement any of these buffers using a 1-of-3 or a 1-of-4 encoding we need to extend the implementation of the state bit to 1-of-3 and 1-of-4 codes. In case of a 1-of-4 code, an alternative solution would be to use two dual-rail state bits internally. However, such a solution would add and extra gate in the pull-down of the inverted output, since decoding the 1-of-4 state would necessitate using both dual-rail bits.

The solution we are suggesting is to generalize the dual rail state bit to 1-of-4 (and 1-of-3) encoding in the following way (for the *standard buffer*):

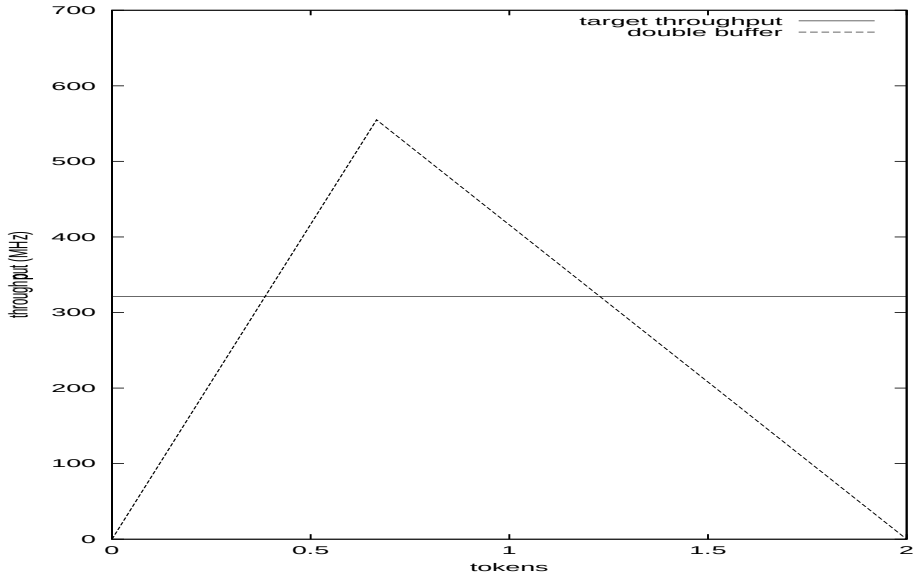


Figure 12: *Dynamic slack in double buffers*

$$
\begin{aligned}
&\_en \ \wedge \ l.0 \ \longrightarrow \ x.1\downarrow \\
&\_en \ \wedge \ l.1 \ \longrightarrow \ x.0\downarrow \\
&\_en \ \wedge \ l.2 \ \longrightarrow \ x.2\downarrow \\
&\_en \ \wedge \ l.3 \ \longrightarrow \ x.3\downarrow \\
&\neg x.1 \ \wedge \ \neg x.2 \ \wedge \ \neg x.3 \ \longrightarrow \ x.0\uparrow \\
&\neg x.0 \ \wedge \ \neg x.2 \ \wedge \ \neg x.3 \ \longrightarrow \ x.1\uparrow \\
&\neg x.0 \ \wedge \ \neg x.1 \ \wedge \ \neg x.3 \ \longrightarrow \ x.2\uparrow \\
&\neg x.0 \ \wedge \ \neg x.1 \ \wedge \ \neg x.2 \ \longrightarrow \ x.3\uparrow
\end{aligned}
$$

$$
\begin{aligned}
&\ l.0 \ \wedge \ x.0 \ \vee \ l.1 \ \wedge \ x.1 \ \longrightarrow \ \_xl1\downarrow \\
&\neg l.0 \ \wedge \ \neg l.1 \ \longrightarrow \ \_xl1\uparrow \\
&\ l.2 \ \wedge \ x.2 \ \vee \ l.3 \ \wedge \ x.3 \ \longrightarrow \ \_xl2\downarrow \\
&\neg l.2 \ \wedge \ \neg l.3 \ \longrightarrow \ \_xl2\uparrow \\
&\neg en \ \wedge \ (\neg \_xl1 \ \vee \ \neg \_xl2) \ \longrightarrow \ \_le\uparrow \\
&en \ \wedge \ \_xl1 \ \wedge \ \_xl2 \ \longrightarrow \ \_le\downarrow
\end{aligned}
$$

The disadvantage of this solution is that the state bits are interfering. By making them noninterfering would add 3 more p-gates in series to the pull-up of the state variable. However, this kind of interfering state could be made

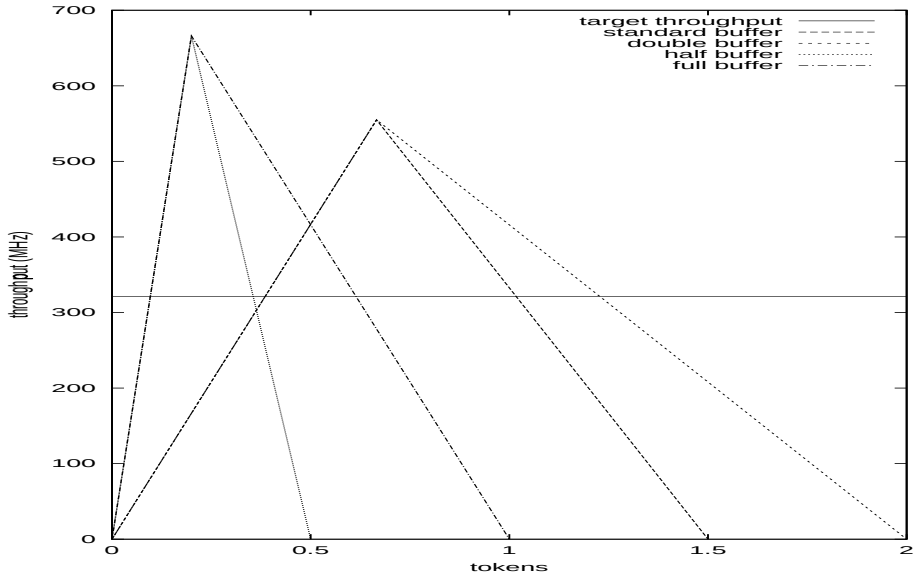


Figure 13: *Comparison between buffers of different slack*

safe easily by ratioing; as a result, dual-rail interfering state bits were used in several places in the Caltech MiniMIPS microprocessor, including the register file.

3.8 Comparison with existing buffers

3.8.1 Slack properties

We already knew that by putting several half-buffers next to each other we can obtain any dynamic slack. However, the point of a high dynamic slack buffer is to render that slack for as low area penalty as possible. The 1-of-4 implementation of the *standard buffer* is about 29% bigger than the 1-of-4 *full buffer* used as a high slack buffer in the Caltech MiniMIPS microprocessor. With the given sizes, the *standard buffer* renders 30% more dynamic slack than the *full buffers* for the same area. The 1-of-4 implementation of the *double buffer* is about 36% bigger than the 1-of-4 *full buffer* and renders 55% more dynamic slack than a *full buffer* for the same area. Figure 13 shows a comparison between the dynamic slacks of different types of buffers.

If one would implement a *bundled double buffer*, (i.e. 8-bit will have a common enable) one can share some of the control circuitry among the

individual quads and reduce the overall size. The unit will no longer be 12 transitions per cycle but would go up to 16. However, even in this case the dynamic slack will be high enough. The same thing cannot be done for the *full buffers* since it heavily relies (for high dynamic slack) on the fact that it run at 10 transitions per cycle.

3.8.2 Logic properties

As to any LR-buffer, one can add logic to the *standard* and *double* buffer. In this way, the buffering property could be combined with the computing capacity. As an example consider an incrementer:

$$inc \equiv *[A?a,\ \ CIN?cin;\ \ S!(a \vee cin),\ \ COUT!(a \wedge cin)]$$

The handshaking expansion for the *standard buffer* reshuffling is:

$$*[([A]; x\Uparrow; Ae\downarrow), ([CIN]; y\Uparrow; CINe\downarrow), ([\neg Se]; S\Downarrow), ([\neg COUTe]; COUT\Downarrow);$$
$$([\neg A]; Ae\uparrow), ([\neg CIN]; CINe\uparrow), ([Se \wedge x \wedge y]; S\Uparrow), ([COUTe \wedge x \wedge y]; COUT\Uparrow)]$$

and the PRs are:

$$\neg a.0 \ \wedge\ \neg a.1 \ \longrightarrow\ \_av\uparrow$$
$$\neg cin.0 \ \wedge\ \neg cin.1 \ \longrightarrow\ \_cinv\uparrow$$
$$Reset\_ \ \wedge\ en \ \wedge\ \_av \ \wedge\ \_cinv \ \longrightarrow\ \_le\downarrow$$
$$\neg\_le \ \longrightarrow\ le\uparrow$$

$$en \ \wedge\ latchout \ \wedge\ s.e \ \wedge\ x.0 \ \wedge\ y.0 \ \longrightarrow\ \_s.0\downarrow$$
$$en \ \wedge\ latchout \ \wedge\ s.e \ \wedge\ (x.1 \ \vee\ y.1) \ \longrightarrow\ \_s.1\downarrow$$
$$\neg\_s.0 \ \vee\ \neg\_s.1 \ \longrightarrow\ sv\uparrow$$
$$\neg\_s.0 \ \longrightarrow\ s.0\uparrow$$
$$\neg\_s.1 \ \longrightarrow\ s.1\uparrow$$

$$en \ \wedge\ latchout \ \wedge\ cout.e \ \wedge\ (x.0 \ \vee\ y.0) \ \longrightarrow\ \_cout.0\downarrow$$
$$en \ \wedge\ latchout \ \wedge\ cout.e \ \wedge\ x.1 \ \wedge\ y.1 \ \longrightarrow\ \_cout.1\downarrow$$
$$\neg\_cout.0 \ \vee\ \neg\_cout.1 \ \longrightarrow\ coutv\uparrow$$
$$\neg\_cout.0 \ \longrightarrow\ cout.0\uparrow$$
$$\neg\_cout.1 \ \longrightarrow\ cout.1\uparrow$$

$$sv \ \wedge \ coutv \ \longrightarrow \ latchout\downarrow$$
$$\neg Reset\_ \ \vee \ \neg latchout \ \wedge \ \neg\_le \ \longrightarrow \ \_en\uparrow$$
$$\_en \ \longrightarrow \ en\downarrow$$

$$Reset \ \vee \ \_en \ \wedge \ a.0 \ \longrightarrow \ x.1\downarrow$$
$$Reset\_ \ \wedge \ \_en \ \wedge \ a.1 \ \longrightarrow \ x.0\downarrow$$
$$\neg x.0 \ \wedge \ (\neg a.0 \ \vee \ \neg\_en) \ \longrightarrow \ x.1\uparrow$$
$$\neg Reset\_ \ \vee \ \neg x.1 \ \wedge \ (\neg a.1 \ \vee \ \neg\_en) \ \longrightarrow \ x.0\uparrow$$
$$a.0 \ \wedge \ x.0 \ \vee \ a.1 \ \wedge \ x.1 \ \longrightarrow \ \_av\downarrow$$

$$Reset \ \vee \ \_en \ \wedge \ cin.0 \ \longrightarrow \ y.1\downarrow$$
$$Reset\_ \ \wedge \ \_en \ \wedge \ cin.1 \ \longrightarrow \ y.0\downarrow$$
$$\neg y.0 \ \wedge \ (\neg cin.0 \ \vee \ \neg\_en) \ \longrightarrow \ y.1\uparrow$$
$$\neg Reset\_ \ \vee \ \neg y.1 \ \wedge \ (\neg cin.1 \ \vee \ \neg\_en) \ \longrightarrow \ y.0\uparrow$$
$$cin.0 \ \wedge \ y.0 \ \vee \ cin.1 \ \wedge \ y.1 \ \longrightarrow \ \_cinv\downarrow$$

$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg\_av \ \wedge \ \neg\_cinv \ \longrightarrow \ \_le\uparrow$$
$$\_le \ \longrightarrow \ le\downarrow$$

$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg s.e \ \longrightarrow \ \_s.0\uparrow$$
$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg s.e \ \longrightarrow \ \_s.1\uparrow$$
$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg cout.e \ \longrightarrow \ \_cout.0\uparrow$$
$$\neg Reset\_ \ \vee \ \neg en \ \wedge \ \neg cout.e \ \longrightarrow \ \_cout.1\uparrow$$

$$\_s.0 \ \wedge \ \_s.1 \ \longrightarrow \ sv\downarrow$$
$$\_s.0 \ \longrightarrow \ s.0\downarrow$$
$$\_s.1 \ \longrightarrow \ s.1\downarrow$$

$$\_cout.0 \ \wedge \ \_cout.1 \ \longrightarrow \ coutv\downarrow$$
$$\_cout.0 \ \longrightarrow \ cout.0\downarrow$$
$$\_cout.1 \ \longrightarrow \ cout.1\downarrow$$

$$\neg sv \ \wedge \ \neg coutv \ \longrightarrow \ latchout\uparrow$$
$$Reset\_ \ \wedge \ latchout \ \wedge \ \_le \ \longrightarrow \ \_en\downarrow$$
$$\neg\_en \ \longrightarrow \ en\uparrow$$

This example shows how the *standard buffer* (or *double buffer*) could be used as a computing stage. The inputs are latched in parallel with reseting the outputs and then, the inputs are reset in parallel with setting the outputs.

The *standard buffer* was used in the implementation of the HiLo register of the Multdiv unit in the Caltech MiniMIPS microprocessor. There are at least two more places in the Caltech MiniMIPS microprocessor where the use of the *standard* or *double* buffer would have resulted in significantly smaller circuits. One of them is the *program counter incrementer* in the Fetch unit and the other is the *exception program counter queue* between the Fetch unit and the CP0 unit.

References

1. Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, J. Staunstrup, ed. North-Holland, 1990.

2. John Hennessey and David Patterson. *Computer Architectures: A Quantitative Approach.* Morgan Kaufmann, 1990.

3. Rajit Manohar. Ph.D. thesis to be published. Caltech 1998

4. Mika Nyström. Master thesis. Caltech unpublished

5. G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice-Hall, 1992

6. Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth and Uri Cummings. *The Design of an Asynchronous MIPS R3000 Microprocessor* 1997

7. Andrew Lines. Master thesis. Caltech unpublished

Acknowledgments

I wish to thank the members of the Asynchronous VLSI Group at Caltech for many stimulating discussions: Andrew Lines, Rajit Manohar, Mika Nyström, Robert Southworth and Catherine Wong. Special thanks to Rajit Manohar for designing and laying out the branch adder and for the very instructive discussions we had, to Uri Cummings for designing and laying out the incrementer and to Andrew Lines for the vehement debates we had over the FETCH.

Last, but not least, thanks to Alain Martin for being my advisor and teaching me asynchronous VLSI.